VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA
EKONOMICKÁ FAKULTA

KATEDRA SYSTÉMOVÉHO INŽENÝRSTVÍ

Vývoj událostně orientované aplikace

Developing an Event Driven Application

Student: Vítězslav Lanc

Vedoucí bakalářské práce: RNDr. Jaroslav Ševčík, Ph.D.

Ostrava 2017

VŠB - Technická univerzita Ostrava
Ekonomická fakulta
Katedra aplikované informatiky

Zadání bakalářské práce

Student:	**Vítězslav Lanc**
Studijní program:	B6209 Systémové inženýrství a informatika
Studijní obor:	6209R017 Informatika v ekonomice
Téma:	Vývoj událostně orientované aplikace Developing an Event Driven Application
Jazyk vypracování:	čeština

Zásady pro vypracování:

1. Úvod
2. Teoretická východiska pro návrh událostmi řízených architektur
3. Analýza současných technik událostně řízeného programování
4. Návrh a implementace aplikace
5. Závěr

Seznam použité literatury
Seznam zkratek
Prohlášení o využití výsledků bakalářské práce
Seznam příloh
Přílohy

Seznam doporučené odborné literatury:

FASION, Ted. *Event-Based Programming: Taking Events to the Limit.* New York: Springer - Verlag New York, Inc., 2006. ISBN 978-1-59059-643-2.
ETZION, Opher and Peter NIBLETT. *Event Processing in Action.* Stamford: Manning Publications Co., 2011. ISBN 9781935182214.
HORSTMANN, Cay S. *Core Java.* 10. vyd. New York – Prentice Hall, 2016. ISBN 978- 0-13-417730-4.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **RNDr. Jaroslav Ševčík, Ph.D.**

Datum zadání: 18.11.2016
Datum odevzdání: 05.05.2017

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA * OSTRAVA Ekonomická fakulta 1

Ing. Petr Rozehnal, Ph.D.
vedoucí katedry

prof. Dr. Ing. Zdeněk Zmeškal
děkan fakulty

Čestné prohlášení

Prohlašuji, že jsem bakalářskou práci na téma „Vývoj událostně orientované aplikace“ vypracoval samostatně s použitím uvedené literatury a pramenů.

V Ostravě, 5. května 2017

Vítězslav Lanc

Obsah

1 Úvod

V dnešní době existuje nemálo přístupů a architektur k vývoji aplikací nebo návrhu informačních struktur. Některé z přístupů k vývoji či psaní kódu mohou být již zastaralé, jiné inovativní, ale není možno říct: „A je nejlepší a B je nejhorší.“ Důležité je rozpoznat, který postup je pro danou situaci nejvhodnější, a kterému se naopak raději vyhnout. Kdyby se snažilo napasovat hrubou silou jeden postup pro řešení veškerých problémů, výsledky by nemohly být uspokojivé.

Zpracování událostí je důležitým prvkem v oblasti informatiky. A nejen informatiky, ale i každodenního života. I pouhé objednání kávy, ať už v kavárně či automatu, je událostí. Člověk dostane chuť na kávu, tak vyhledá automat a objedná si požadovanou kávu. Tím vzniká událost, kterou je třeba zpracovat, aby to nedopadlo tak, že zákazník dostane místo kávy čaj. U tak jednoduchého příkladu se to může zdát jako drobnost, ale při narůstající složitosti požadavků a systémů, které je zpracovávají, roste také důležitost správného přístupu k zpracování událostí.

Díky správně navržené architektuře je umožněno bezproblémové rozšiřování stávajícího stavu a také je zajištěn efektivní chod aplikace, dokonce něčeho většího, třeba celého systému. Událostně orientované architektury jsou zaměřeny na zpracování událostí a zajištění vhodné reakce. Tyto architektury sice nejsou žádnou novinkou, ale míra jejich využití stoupá. Především v posledních letech.

Cílem této práce je analyzovat současné přístupy tvorby událostně orientovaných architektur, popis návrhu těchto architektur a využití získaných poznatků při návrhu a implementaci událostně procesní sítě vlastní aplikace.

Cílem druhé kapitoly je vytvořit pevný a jasný teoretický základ pro pochopení událostního řízení. V kapitole jsou rozebrány základy událostního řízení v prostředí reálného světa a možné postoje k dané problematice. Cílem třetí kapitoly je analyzovat současné techniky tvorby událostně procesních sítí. Je vysvětleno, jak jsou událostně orientované architektury implementovány a jak jsou události zpracovány. Cílem čtvrté kapitoly je přiblížit, jak byly teoretické poznatky využity při vývoji vlastní aplikace. Dále jsou popsány stěžejní postupy při její tvorbě, rozbor vrstev, bloků aplikace a hlavních tříd.

2 Teoretická východiska pro návrh událostmi řízených architektur

2.1 Úvod do událostního řízení

V každodenním životě je člověk vystaven nepřetržitému sledu příhod, které vědomě či podvědomě zpracovává. Každou sekundu je lidský mozek bombardován obrovským množstvím informací. Některé se objeví ve stejný okamžik, jiné postupně. U dalších se jejich význam neprojeví hned a porozumět mu bude možno až s odstupem času po pochopení dalších souvislostí (Alves, et al., 2013). Kupříkladu, když jsou člověkem zaznamenány vjemy jako: zvuk výstřelu, křik lidí, utíkající lidé a radostný jásot, samy o sobě nemají význam a nemůže být přesně řečeno, s čím dané vjemy souvisí. Teprve když se spojí dohromady, může být pochopen kontext a souvislosti mezi jednotlivými událostmi. Lidský mozek je mocný nástroj, kterým je umožněno spojování těchto souvislostí (Alves, et al., 2013).

V softwarových systémech je pojem událost již slušnou dobu, i když někdy pod jinými jmény (Fasion, 2006). Moderní událostní řízení (*Event Processing*) se stává důležitou částí informačního odvětví. Technika se objevila koncem 90. let, ale tento koncept není ničím novým. Již po 60 let byl a stále je technologickým základem pro diskrétní událostní simulace, sítě a internet, simulace počasí a předpovědi a mnohé další (Luckham, 2011).

Událostní řízení je neustále rozrůstající se odvětví, zabývající se softwarovými systémy založenými na událostech, což dále zahrnuje filtrování, transformování, nebo nalézání vzorů, ve kterých se události vyskytují či vznikají. Podle některých studií je událostní řízení považováno za rychle rostoucí odvětví (Etzion & Niblett, 2011).

Událostně procesní architektura dovede zpracovat rostoucí počet informací z velkého množství zdrojů, okamžitě pochopit jejich souvislosti s událostmi odehrávající se teď nebo v blízké budoucnosti. Tyto vlastnosti jsou označovány jako *Real-Time Situation* (Alves, et al., 2013). Takové systémy zajišťují uživateli mobilitu a poskytují relevantní informace v reálném čase. Systémy jsou velmi flexibilní při zpracování a reagování na vyskytující se události. Tyto systémy jsou užívány v mnoha odvětvích, jako zdravotní péče, doprava, vzdělání a mnohé další (Babei, et al., 2016).

Jak je možno vypozorovat, události sehrávají důležitou roli v každodenním životě. Událostní řízení je rozsáhlým tématem a v následujících částech práce jsou popsány základy a terminologie, popis událostně procesních sítí, jednotlivých elementů a principů, na kterých fungují. Událostní řízení je také více popsáno v kontextu s každodenním životem.

2.2 Událost

Veškeré událostní řízení je založeno na velmi podstatné entitě – události. Je důležité, aby bylo ujasněno, jak je třeba v kontextu této práce na takovou událost nahlížet.

Definice: ***Událost*** *je zaznamenatelný stav, který může spustit patřičnou reakci* (Fasion, 2006).

Definice: ***Událost*** *je jev, který se vyskytne v rámci určitého systému. Tento jev již proběhl, nebo právě probíhá. Výraz „událost“ je také používán jako pojmenování programové entity reprezentující daný výskyt události ve výpočetním systému* (Etzion & Niblett, 2011).

Událostní řízení se zabývá událostmi, které se vyskytnou a odehrají v reálném světě, jako příjezd vlaku nebo objednání jídla. Techniky událostního řízení je možno použít i v počítačových oblastech, jako například trénovací simulátory, virtuální realita a další (Etzion & Niblett, 2011).

2.3 Událostmi řízené chování v denním životě

Koncept událostí je jednoduchý a přesto mocný. Pro ilustraci je představen následující příklad. Člověk sedí v bance a vyčkává, až na něj přijde řada. Od chvíle, kdy vešel, se stalo několik věcí: vytáhl si z automatu lístek s číslem, udávající jeho pořadí, lidé se v klidu vystřídali na jednotlivých přepážkách a mnohé další, nijak zvlášť poutavé události. Tato pokojná atmosféra by mohla být narušena tím, že do banky vtrhnou dva lupiči. Lidé by byli v šoku a museli by na tuto událost patřičně reagovat. Jeden by přistoupil k pobočce se zbraní a nutil pracovníka banky o vydání peněz.

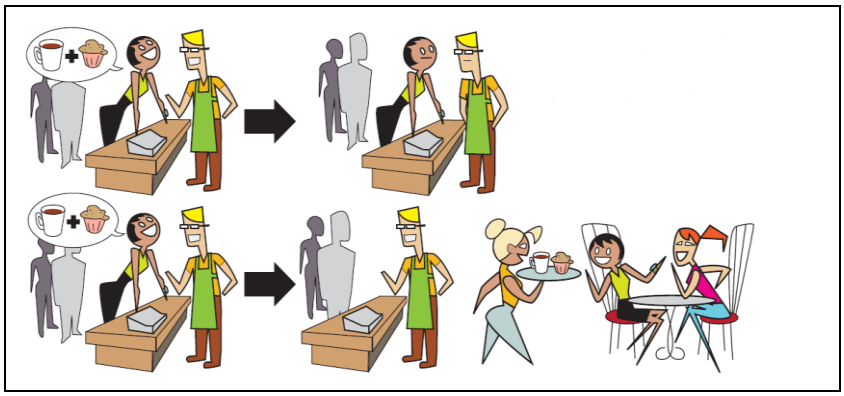
Druhý by dával pozor na zákazníky, aby například nezavolali policii, což by vyvolalo další sled událostí. Jak je možné vypozorovat, jedna událost může vyvolat mnoho dalších, které mají různý charakter.

2.3.1 Synchronní a asynchronní přístup

Interakce může mít synchronní, nebo asynchronní charakter. Při synchronní interakce jsou očekávány rychlé reakce v řádu desetin sekund. Entita, která formulovala požadavek čeká na odpověď a nedělá nic jiného. V případě asynchronní komunikace však může v době čekání provádět další akce (Etzion & Niblett, 2011).

Jednoduchým příkladem po vysvětlení této vlastnosti je telefonní komunikace. Telefonní hovor je synchronní forma komunikace. Volající může komunikovat s volaným pouze tehdy, je-li volaný v danou chvíli dostupný. Telefonní záznamník však umožňuje komunikaci asynchronní. Pokud volaný není dostupný, volající může zanechat zprávu a poté telefonát ukončit, přestože se mu v danou chvíli nedostalo odpovědi. Až se to volanému bude hodit, poslechne si zprávu. Asynchronní komunikace je v tomto případě pro jednotlivce jednodušší (Hohpe & Woolf, 2011). Většina prvků digitálního světa pracuje v asynchronním režimu a kdykoliv potřebují vzájemně sdělovat informace, je nutno jejich komunikaci synchronizovat (Miskowicz, 2015). Skutečný svět je často asynchronní. Každodenní život se skládá z mnoha koordinovaných asynchronních interakcí, což vede k tomu, že asynchronní komunikační architektura může být přirozenou cestou při modelování těchto interakcí (Hohpe, 2005).

Díky těmto vlastnostem je možno přistupovat k řízení událostí různými způsoby. Pro ilustraci je představen příklad z kavárny. Synchronní přístup: zákazník přijde k pultu, objedná si kávu a pečivo. Osoba za pokladnou dá zapéct pečivo do trouby, pak přichystá kávu, vyndá pečivo z trouby, přijme peníze a předá objednávku zákazníkovi. Dalším způsobem, jak mohou být objednávky vyřizovány, je za pomocí asynchronních principů. Situace by mohla být řešena tak, že za pultem by byla osoba vyřizující objednávky a platby. Objednávky by nahlásila dalším zaměstnancům, kteří by tyto objednávky připravovali. Osoba za pultem by tak po úspěšném uzavření objednávky mohla hned obsluhovat další osobu, zatímco první zákazník by si již mohl jít sednout a čekat, až mu objednávku přinese obsluha (Etzion & Niblett, 2011). Kontrast těchto přístupů je zobrazen na obr. 2.1.



Obrázek 2.1 Ukázka synchronního a asynchronního přístupu (Etzion & Niblett, 2011)

2.3.2 Různé úrovně událostí

Na samotné události může být nahlíženo z různých pohledů, mohou být seskupovány například dle typu. Typ událostí je více rozebrán v třetí kapitole této práce. Pro začátek stačí vědět, že různý pohled na události umožňuje rozlišovat události vyšší či nižší úrovně, které mohou představovat různé úrovně abstrakce, které jsou popsány v pozdější části této kapitoly. Události nižší úrovně jsou jednoduše zpozorovatelné, kdežto události na vyšších úrovních vyžadují větší míru pozornosti.

Události je možné detekovat každý den. Některé je možno zpozorovat snadno, jako věci, které vidíme a slyšíme během dne. Jiné se vyskytnou, pokud jsou před tím splněny jisté podmínky. Například: aby mohl člověk pravidelně obdržet časopis, musí si jej nejdříve předplatit. Aby bylo možno zpozorovat další typ událostí, je nutno provést jistá opatření nebo zkoumání, aby bylo možno s jistotou určit úroveň dané události. Kupříkladu: aby si člověk uvědomil, že jeho konzumace mléka byla zvýšena, nestačí pouze přijít k lednici a zjistit, že mléko došlo. Je nutno vybavit si, jak dlouho průběžně trvalo spotřebovat krabici mléka před měsícem, a porovnat s tím, jak dlouho to trvá teď. V tomto případě je „došlo mléko“ jednoduše zpozorovatelná událost, kdežto „spotřeba mléka byla zvýšena“ je událost na vyšší úrovni a je třeba ji vydedukovat pozorováním událostí na nižší úrovni (Etzion & Niblett, 2011).

Hlavním důvodem monitorování výskytu událostí, je příležitost k reakci. Možnou reakcí na událost v předchozím příkladu je zvýšení frekvence nákupu mléka. Mnoho událostí vyskytujících se každý den nejsou nijak podstatné. Mohou být pouze šumem v pozadí, který nevyžaduje žádnou reakci. Některé události ale reakci vyžadují. Jednou z hlavních oblastí událostního řízení je detekce událostí a hlášení jejich výskytů. Pak je na ně možno adekvátně zareagovat (Etzion & Niblett, 2011).

2.4 Událostní řízení

V předchozích částech byly situace představovány hlavně na příkladech z běžného života. V následujících částech je představeno řízení událostí v oblasti informatiky.

Událostní řízení získalo značnou pozornost jako samostatná disciplína informatiky (Bruns & Dunkel, 2014). Jedná se o soubor technik a nástrojů, pomáhající pochopit a řídit událostmi řízené informační systémy. Samotné události mohou být řazeny do kategorií, například podle místa vzniku, příčiny nebo času vzniku. Tyto vztahy přidávají nové možnosti řízení informačních systémů. Je důležité, aby řízení těchto systému bylo co nejkvalitnější, jelikož dnešní informační společnost je založena na sběru a sdílení informací. Nejen komerční, ale i vládní nebo armádní odvětví jsou závislé na zpracování těchto informací. Naše společnost je v dnešní době na informačních technologiích dosti závislá (Luckham, 2001).

Přes dobu vývoje samotné techniky bylo jejich principů užíváno například v:

- diskrétních událostních simulacích,
- sítích,
- aktivních databázích,
- servisně a událostně řízených architekturách.

S nejvyšší pravděpodobností se vývoj této techniky nezastavil a bude pokračovat i v budoucnu (Luckham, 2012).

Definice: ***Událostní řízení*** *je soubor technik a nástrojů provádějící operace s událostmi. Základní operace událostního řízení zahrnují čtení, vytváření, transformace a mazání událostí* (Etzion & Niblett, 2011).

Událostní řízení zahrnuje dvě hlavní odvětví:

- **Návrh a kódování** aplikací užívajících, ať už přímo nebo nepřímo, událostí. Často označováno jako událostně založené programování, událostmi řízené architektury nebo událostně procesní platformy.
- **Zpracování operací**, které je možno takovou aplikací provádět. Operace zahrnují filtrování určitých událostí, změny událostí, zkoumání souhrnu událostí za účelem nalezení vzoru (Etzion & Niblett, 2011).

2.5 Hodnota událostního řízení v kontextu podniku

Událostní řízení je rozšiřující se technikou umožňující získání relevantních informací z distribuovaných systému v reálném nebo téměř reálném čase *(viz Úvod do událostního řízení)*. Na technologii je směřováno čím dál tím více pozornosti v souvislosti se servisně orientovanými architekturami. Podpora událostního řízení je vnímána jako kritický faktor úspěchu mnoha podniků (Paschke, 2009).

V dnešní době je v informatice jedním z problémů obrovský růst objemu dat. Informace přichází z mnoha různých systémů, v ohromných množstvích, v různém čase, některé s vyšší a jiné s nižší prioritou. Hlavními důvody růstu objemu jsou:

- klesající náklady na výpočetní techniku,
- nárůst síťových kapacit,
- „*Big Data*[1]“,
- úspěšnost servisně orientované architektury[2], vedoucí k užití principu opakované použitelnosti,
- rostoucí poptávka podniků po informačních technologiích (Alves, et al., 2013).

[1] „*Big Data*“ jsou charakterizována objemem, různorodostí, rychlostí a přesností (shodou se skutečností). „*Big Data*“ jsou rozšiřující se oblastí získávající pozornost inženýrů při vývoji nových generací důmyslně propojených zařízení. Také v oblasti zdravotnictví se rozšiřuje možnost přístupu k informacím odkudkoliv na světě (Bhatt, et al., 2017). „*Big Data*“ se nejčastěji vyskytují v nestrukturovaných formátech, většinou získána ze sociálních sítí, telefonů, mobilních telefonů a dalších zdrojů (Alves, et al., 2013).

[2] Servisně orientované architektury jsou založeny na definici rozhraní a staví na nich svou celou topologii, jejich implementaci a komunikaci (Binildas, et al., 2008). Hlavním znakem servisně orientovaných architektur je to, že procesy a aplikace založené na této architektuře nejsou navrhovány jako komplexní programové struktury, ale řízeny jako nezávislé oddělené servisní jednotky. Je kladen důraz na opakovatelnou použitelnost navrhovaných komponent. Díky toho vznikají flexibilní systémy, snadno přizpůsobitelné novým podmínkám (Schmutz, et al., 2010).

Při neustále rostoucí komplexnosti těchto systémů a růstu objemu informací rostou také požadavky na komplexnost řízení těchto systémů. Událostmi řízené architektury a systémy jsou sofistikované a vnitřně vybavené pro řešení těchto problémů, či dokonce řízení celých podniků, díky tomu, že události mohou být okamžitě zpracovány a interpretovány (Alves, et al., 2013).

2.5.1 Událostně procesní platformy

Konkrétnějšímu popisu událostně procesních sítí je věnována třetí kapitolu této práce, je ale třeba uvést základní principy, aby bylo možno popsat využití těchto platforem v podniku.

Je možno psát událostně založené programy bez užití událostně procesních operací. Jsou ale vlastnosti odlišující událostně procesní platformy od obyčejných programů.

- Abstrakce – Operace formující logiku řízení událostí mohou být odděleny od aplikační logiky, což umožňuje, že mohou být změněny, aniž by musela být měněna celá aplikace.
- Oddělenost – Události zpozorované a vyprodukované jednou aplikací mohou být přijaty ke zpracování úplně jinou aplikací. Není nutno, aby o sobě aplikace navzájem věděly. Událost může být přijata a zpracována mnoha aplikacemi.
- Zaměřenost na reálný svět – Událostně procesní platformy se často zabývají událostmi, které již proběhly nebo se právě odehrávají v reálném světě (Etzion & Niblett, 2011).

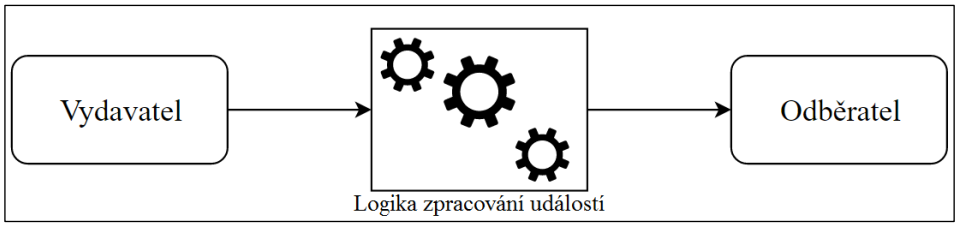
2.5.2 Důvody pro použití událostně procesních platforem

Událostně procesní platformy umožňují flexibilní rozšíření existující aplikace. Místo toho, aby se měnila ta původní kvůli přidání pár funkcí navíc, je jednodušší pouze přidat další komponentu produkující události související s požadovanými funkcemi. Tyto platformy dále umožňují asynchronní přístup, což se velice hodí tam, kde se události vyskytují nepravidelně a nepředvídatelně.

V platformách soustředěné na událostně procesní architektury je logika zpracování událostí oddělována jak od části, která události produkuje, tak od části,

která události přijímá[3]. Tato oddělenost je zobrazena na obr. 2.2. Díky tomu je možno komponentu obsahující událostně procesní logikou použít kdekoliv, kde je užito událostně procesní platformy. Tyto platformy poskytují některé, nebo všechny z následujících:

- jazyk pro tvorbu událostně procesní logiky,
- nástroje pro návrh a testování událostně procesní logiky,
- prostředí pro zpracování událostně procesní logiky,
- událostně distribuční mechanismy (Etzion & Niblett, 2011).



Obrázek 2.2 Struktura událostně procesní aplikace

Jednou z vlastností událostně procesních platforem je abstrakce. Náklady na tvorbu systémů mohou být díky této abstrakci sníženy, podobně, jako když je pro uchovávání dat použito databázových systémů místo systému souborů. Událostně procesní software poskytuje nástroje pro správu událostí, jež jsou na vyšší úrovni než ty z pohledu programovacích jazyků *(viz Různé úrovně událostí)*. Díky tomu dochází k snížení nákladů spojenými s vývojem a údržbou. Další výhodou je agilnost. Změna událostně procesních funkcionalit může být podnikem provedena relativně rychle, díky tomu, že události s vyšší úrovní abstrakce jsou odděleny od hlavní aplikační logiky. V některých situacích rostoucí objem a složitost událostí zvyšuje požadavky na výkon, spolehlivost, dostupnost a bezpečnost. Optimalizovaný software na řešení těchto problémů může být kritický k dosažení těchto cílů, což u obyčejného programování může být problém. Naopak v jiných případech, kdy struktura podniku vyžaduje pouze jednoduché zpracování událostí, nemusí být vhodné investovat čas a peníze na implementaci událostně procesních platforem (Etzion & Niblett, 2011).

[3] V práci je nadále komponenta produkující události označován jako „vydavatel“ a komponenta přijímající události ke zpracování „odběratel“. Vydavatel i odběratel budou podrobněji popsány v třetí kapitole této práce, prozatím stačí vědět, že vydavatel je entita, která události nevytváří, ale poskytuje události událostně řídícímu systému. Odběratel je entita tyto události přijímající.

Není možno říct, že událostní řízení je univerzální řešení. Událostní řízení však hraje důležitou roli. Nejde o to, aby bylo na této technologii postaveno vše, ale často je to podstatná doplňující část (Etzion & Niblett, 2011).

2.6 Propojení událostního řízení se souvisejícími koncepty

V závěrečné části této kapitoly je popsáno, jak je možno využít událostně procesních architektur v kontextu řízení podniku.

2.6.1 Událostně orientované řízení podnikových procesů

Správné řízení podniků je kritický důležité pro zajištění jeho správného chodu. Je proto důležité, aby pro řízení podnikových procesů byl zvolen správný přístup.

Definice: ***Podnikový proces*** *se skládá z činností prováděné podnikem v organizačním a technickém prostředí. Tyto činnosti společně realizují cíle daného podniku. Každý proces se odehrává v prostředí jedné organizace, ale může komunikovat s procesy prováděnými jinými podniky* (Weske, 2007).

Definice: ***Řízení podnikových procesů*** *zahrnuje techniky, metody a koncepty pro podporu návrhu, administrace, konfigurace a analýzy procesů podniku* (Weske, 2007).

Řízení podnikových procesů se zabývá počítačovou podporou pro modelování, řízení, uspořádání a vykonávání některých nebo všech podnikových procesů. Synergie mezi řízením podnikových procesů a událostním řízením jsou následující:

- Systémy pro řízení podnikových procesů mohou sloužit jako vydavatel, generující události ohlašující změny v těchto systémech, které jsou následně analyzovány událostně procesní platformou. Vyhodnocené události jsou pak buďto vráceny systému pro řízení procesů, nebo jsou dále šířeny do jiných aplikací.
- Systémy pro řízení procesů mohou sloužit jako odběratel událostí reagující na události, jež byly zaznamenány a zpracovány událostně procesní platformou.

Událost by následně mohla například spustit nový proces, změnit běh existujícího procesu, nebo daný proces zastavit (Etzion & Niblett, 2011).

2.6.2 Monitorování podnikových aktivit

Monitorování podnikových aktivit je řešení nabízející sledování podnikových událostí v reálném čase, což také zahrnuje sledování podnikových procesů, operačních aktivit a různých situací, ve kterých se podnik nachází. Podnikové události řídící monitorování aktivit mohou být zaslány různými aplikacemi nebo technologiemi (Amnajmongkol, et al., 2008). Systém pro monitorování podnikových aktivit může vyžadovat funkcionality událostního řízení, kterými je umožněno plnohodnotnější monitorování například nalézáním vzorů, ve kterých se události vyskytují. Je to také užitečné v situaci, kdy je nutno sledovat data z více zdrojů (Etzion & Niblett, 2011).

2.6.3 Využití událostní řízení pro analytické účely

Organizacemi jsou využívány analytické techniky, často označovány jako „*Business Intelligence*“ (BI), a software pomáhající činit rozhodnutí na základě sesbíraných dat. Dnešní systémy BI se od událostních platforem liší v tom, že jsou řízeny požadavky. Vstupy systému jsou soubory dat, která byla dříve získána z datového skladu. Následně systémy tato data analyzují. Na rozdíl od událostních platforem však nereagují na samotné události v momentě jejich vzniku. Je ale možno použít software poskytující online analýzu pro podporu rozhodování, což je ve své podstatě událostně orientovaný BI, protože zahrnuje online analýzy, které mohou být spuštěny událostmi. Občas se pro takové označení používá termín „*Operational Intelligence*“ (Etzion & Niblett, 2011).

2.7 Shrnutí

V této kapitole je popsán obecný základ událostního řízení. Úvodem kapitoly byl představen koncept událostí a nastíněny jeho souvislosti jak v každodenním životě, tak v oblasti informatiky. Dále bylo vysvětleno, jak jsou události řazeny úrovní na základě úrovně jejich abstrakce. Byly popsány fundamentální principy událostního řízení a možné využití událostně procesních platforem v kontextu řízení podniku.

3 Analýza současných technik událostně řízeného programování

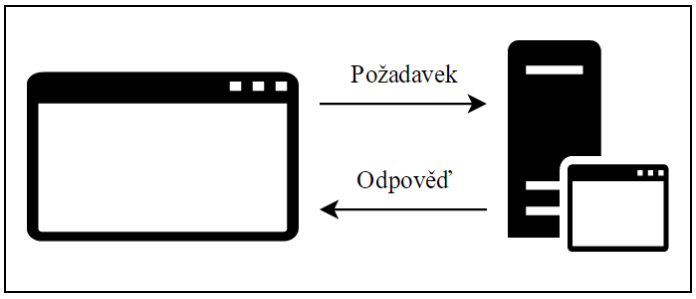
K tvorbě událostně procesních platforem je možno využít více přístupů. Tvorba těchto platforem je označována jako událostně řízené programování. V této kapitole jsou popsány možné přístupy k návrhu takových platforem.

3.1 Událostně řízené programování

Díky správně navržené událostně orientované architektuře je možno redukovat závislost jednotlivých komponent, což je také důvodem, proč je v jistých situacích využito právě událostně řízeného programování. Tyto komponenty pak mezi sebou mohou efektivně komunikovat (Fasion, 2006).

3.1.1 Komunikace požadavek/odpověď v kontrastu s událostmi

Každý uživatel webového prohlížeče má zkušenost s interakcí fungující na principu požadavek/odpověď (obr. 3.1). Uživatelem je formulován požadavek kliknutím na odkaz, nebo vyplněním formuláře, který je odeslán na server. Následně čeká na vygenerování odpovědi. Požadavky mohou být dotazy (*query*), nebo to může být vyjádření žádosti, aby se něco aktualizovalo (*update*). Aktualizace mohou být cokoliv, co způsobí změnu stavu na straně serveru, například přidání zboží do košíku. Požadavky vrací požadované informace, aktualizace většinou potvrzení, že požadovaná akce proběhla v pořádku. Vzor interakce požadavek/odpověď je technika využívána také v programování. Když je program navrhován, může nastat situace, ve které jsou potřebné dodatečné informace. Nejpřirozenější by v takovém případě bylo zavolat externí komponentu nebo službu, která požadovanou informaci vrátí, nebo provede požadovanou akci (Etzion & Niblett, 2011).



Obrázek 3.1 Interakce požadavek/odpověď

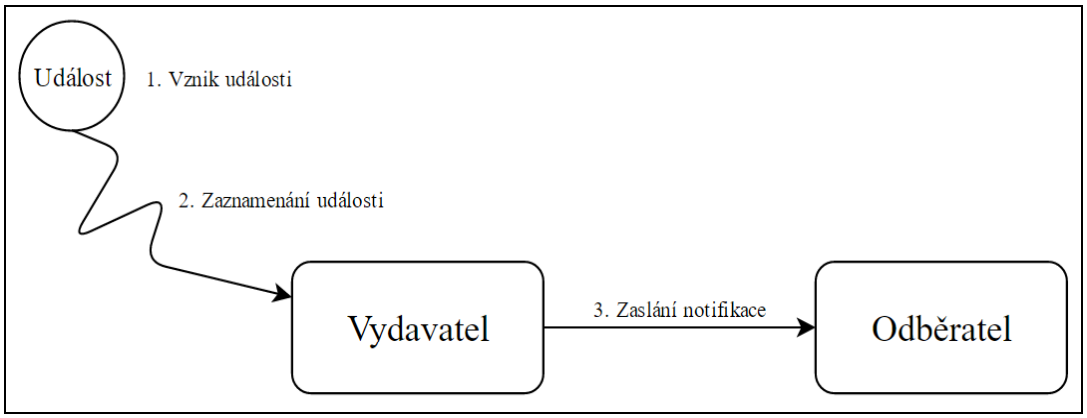
Rozdíl mezi událostí a požadavkem je ten, že událost je projev něčeho, co se již odehrálo, zatímco požadavek vyjadřuje přání, aby se něco teprve odehrálo. V kontextu událostí komunikaci zajišťují vydavatelé a odběratelé událostí, jež byli popsáni v předcházející kapitole *(viz Důvody pro použití událostně procesních platforem).* Rozdíl je demonstrován na následujícím příkladu. V dnešní moderní době má spousta lidí chytrý telefon s přístupem k internetu. Není proto problém si rychle vyhledat například vlaky, které vyjíždí z požadované stanice v určitém časovém rozmezí. Pro tyto účely existují i přímo uzpůsobené aplikace s uživatelským rozhraním, kterým je umožněno pohodlné formulování požadavku. V tomto případě je „vyhledat odjezdy vlaků ze stanice Ostrava – Stodolní od 14:00 až 14:30“ požadavek a „vlak vyjel ze stanice Ostrava – Stodolní 14:25“ je již událost (Etzion & Niblett, 2011).

3.1.2 Princip oddělenosti

Událost má sama o sobě význam, který je nezávislý na vydavatelích a odběratelích událostí, což vede k tomu, že vydavatelé mohou být úplně odděleni od odběratelů, což je další rozdíl oproti interakci požadavek/odpověď, kde daný požadavek má většinou smysl jen v kontextu dané interakce. Samozřejmě že i v této interakci je jistý stupeň oddělenosti. Když je implementován poskytovatel služeb, je navrhován bez ohledu na žadatele služeb. Žadatel služeb je však závislý na poskytovateli tím, že očekává požadovanou akci. Vydavatel událostí není závislý na akcích odběratele, stejně tak odběratel je komponenta nezávislá na odběrateli. Jedinou závislostí na vydavateli je samotné vyprodukování události (Etzion & Niblett, 2011). Odběratel však není povinen nějakou akci provést. A to, jestli je nějaká akce odběratelem provedena, již není starostí vydavatele (Fasion, 2006).

3.1.3 Notifikace

V práci je popsáno zasílání události vydavatelem, odebírání událostí odběratelem. Může být poněkud matoucí, co je tím ve skutečnosti myšleno. Nejedná se o zaslání samotné instance třídy, ale událostí vyvolaný signál nesoucí informace o dané události označovaný jako notifikace. Událost je entitou představující samotný výskyt události, kdežto notifikace je mechanismem pro výměnu této informace (Fasion, 2006).



Obrázek 3.2 Zaslání notifikace

Jak je možno vidět na obr. 3.2, není zasílána samotná událost. V momentě, kdy událost nastane, je na vydavateli, aby vytvořil k dané události notifikaci a tu nadále rozeslal směrem k odběratelům.

3.2 Distribuce událostí

Odběr událostí je proces provázání vydavatele událostí k odběrateli. V podstatě se jedná o rozhodnutí přijímání v budoucnu produkovaných notifikací specifickým vydavatelem. Vydavatelé událostí jsou obvykle schopní detekovat několik typů událostí a zasílat různé typy notifikací. Během procesu odběru musí odběratel určit, o který typ událostí má zájem. Vydavatelem je nadále uložen záznam o preferenci odběratele. V okamžiku, kdy je jím detekována událost, je odběratelům přihlášeným k danému typu událostí zaslána notifikace. Odběratel obvykle může být odebíráno více typů událostí od stejného vydavatele, vydavatelem může být zasílán stejný typ více odběratelům (Fasion, 2006).

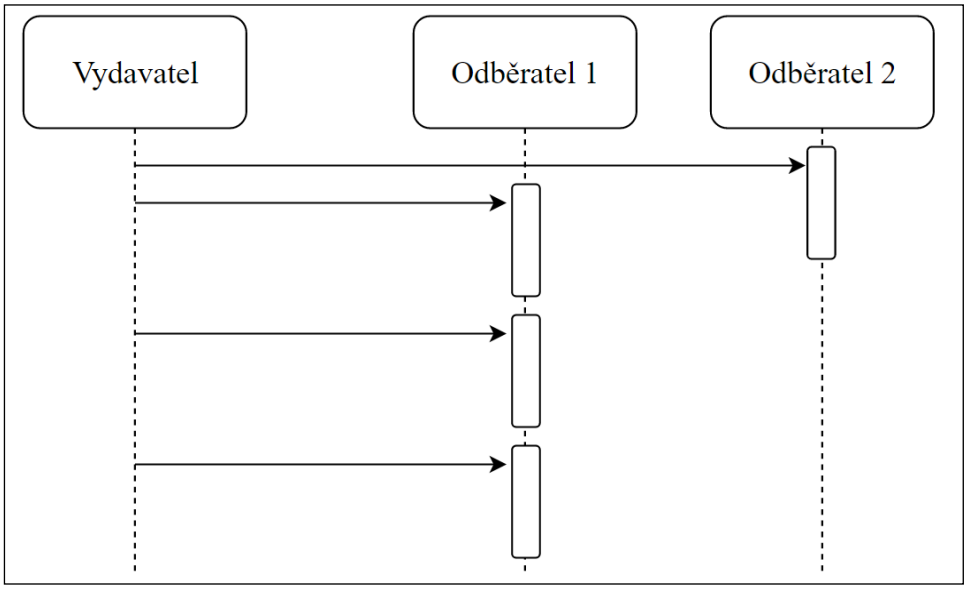
3.2.1 Způsoby interakce

Interakce užívaná událostmi se od interakce požadavek/odpověď jisté odlišnosti, dvě nejvýznamnější jsou následující:

- Vydavatel událostí obecně neočekává, že odběratel provede nějakou specifickou akci, když obdrží událost zaslanou vydavatelem.
- Události jsou často zaslány jako „jednosměrné“ zprávy. Poté, co vydavatel zaslal notifikaci, může pokračovat v dalších věcech, aniž by musel čekat na odpověď (Etzion & Niblett, 2011).

Spouštění událostí

Na obr. 3.3 je zobrazena interakce někdy označována jako tzv. „*push*“ interakce, kde je komunikace iniciována vydavatelem v momentě, kdy existuje událost, která jím může být distribuována. Vydavatelem je zasílána událost každému odběrateli jako jednostranná zpráva.



Obrázek 3.3 Zasílání spouštěcích událostí

V případě zobrazeném na obr. 3.3 je zaslána kopie událostí dvěma odlišným odběratelům, následně dvě další, ale pouze prvnímu. V žádné z těchto interakcí není vydavatelem očekávána odpověď. Prvním z důvodů je ten, že v opačném případě by odpověď byla svázána s reakcí odběratele, ale jak již bylo zmíněno, vydavatel neví, co se bude následně odehrávat. Dalším důvodem je ten, že při návrhu vydavatele událostí často není známý počet odběratelů. Tento počet může být při běhu aplikace nebo událostmi řízeného systému dynamicky měněn. Z obr. 3.3 je dále možno vypozorovat tři běžné vlastnosti distribuce událostí:

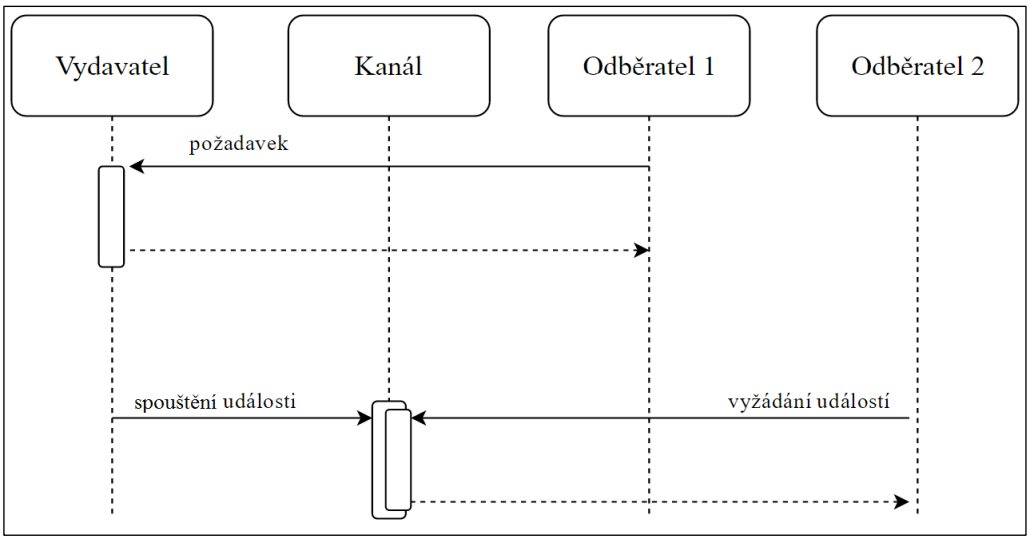
- Odběratelem není zasílána odpověď vydavateli, může však být zaslána indikace, že událost byla obdržena.
- Událost může být zaslána více než jednomu odběrateli a každý ji může zpracovat jiným způsobem.
- Vydavatel událostí může vyprodukovat posloupnost událostí (Etzion & Niblett, 2011).

Fasion (2006) označuje tento druh interakce jako „spouštění událostí“ (*firing events*). Tento typ interakce je jedním z typů interakce použitých při řešení praktické části této práce.

Použitím této interakce může být docíleno snížení procesní latence, protože vydavatelem může být zaslána notifikace ihned v momentě výskytu události. Toto neplatí u interakce popsané v následující části kapitoly, někdy označovanou jako tzv. *„pull“* (Etzion & Niblett, 2011).

Distribuce událostí stylem požadavek/odpověď

V předchozích částech kapitoly byla rozebrána interakce požadavek/odpověď. V této části jsou představeny situace, ve kterých je tato interakce používána k distribuci událostí. V takových případech je událost předána jako parametr buďto požadavku nebo odpovědi. Tento typ interakce může být kombinován s již zmíněným spouštěním událostí, jak je zobrazen na obr. 3.4.



Obrázek 3.4 Kombinace interakcí

Odběratelem událostí je užíván vzor požadavek/odpověď, když žádá událost od vydavatele, nebo od zprostředkovatele, který je popsán v následující části práce. V rámci konkrétní komunikace obdrží odpověď s danou notifikací. Aby bylo zabráněno dlouhému čekání na notifikace, nebo nutnosti vydavatele obsluhovat požadavky mnoha odběratelů, může být delegována část zodpovědnosti na již zmíněného zprostředkovatele (Etzion & Niblett, 2011).

Přestože je obvykle preferováno spouštění událostí, jsou situace, kdy je vhodnější distribuce událostí stylem požadavek/odpověď:

- Odběratel je dostupný jen příležitostně.
- Odběratelem je vyžadována kontrola nad tím, kdy jsou události přijímány.
- Odběratelem není povoleno přijímat příchozí události, například v případě systému chráněného firewallem.
- Vydavatel není schopen distribuovat události.

Běžným příkladem posledního bodu je zapisování událostí do logu. Log je soubor nebo databázový systém používaný pro uchování záznamu o událostech po delší dobu. Daným logem, který je zároveň vydavatelem, události nejsou distribuovány. Je jím však poskytováno rozhraní umožňující číst tyto události (Etzion & Niblett, 2011). Podobného principu je využito v praktické části ke zpracování specifického druhu událostí.

3.3 Událostně procesní a servisně orientované architektury

V posledních letech jsou čím dál častěji užívány termíny událostmi řízené architektury a servisně orientované architektury. To, že oba termíny končí slovem „architektury“, neznamená, že musí být rozhodnuto buď pro jednu, nebo pro druhou. Ve skutečnosti se tyto architektury můžou bez problému doplňovat.

Definice: ***Událostně řízené programování****, také známé jako* ***událostně řízené architektury*** *(Event-driven Architecture) je architektonický styl, ve kterém jedna nebo více komponent v softwarovém systému zpracovává příchozí notifikace* (Etzion & Niblett, 2011).

Definice servisně orientovaných architektur byla zmíněna v dřívějších částech práce (*viz Hodnota událostního řízení*). Pro připomenutí: hlavním užitím servisně orientovaných architektur je návrh aplikací tak, aby byla zachována možnost opakovatelné použitelnosti komponent. Aby toho mohlo být dosaženo, je nutné, aby měly komponenty jasně definovaná rozhraní, která jsou nezávislá na svých implementacích a komponenty musí být možné použít více než v jedné aplikaci. Důraz je kladen na flexibilitu, která z opakované použitelnosti plyne. Služby jsou poskytovány formou požadavek/odpověď. Je samozřejmě možné, aby komponenty

implementovaly oba přístupy. Komponentou tedy mohou být požadavky obsluhovány interakcí požadavek/odpověď a zároveň může být vydavatelem nebo odběratelem událostí. Kupříkladu: komponenta umožňující vyřizování objednávek přijatých jako požadavky, může vyprodukovat notifikaci, když zpozoruje, že na skladě dochází zásoby (Etzion & Niblett, 2011).

3.4 Hlavní pojmy událostně řízených architektur

Do tohoto bodu práce byly vysvětlovány fundamentální principy související s řízením událostí, poukazován výskyt událostí v každodenním životě a základní užití událostního řízení ve světě informatiky a podniků. Dále bylo popsáno, na jakých principech systémy řízené událostmi fungují. Pro ilustraci byly uváděny konkrétní příklady, aby se dalo lépe pochopit, jak takové systémy fungují. Přesto však bylo vše popisováno na poměrně abstraktní úrovni. V následujících částech je již práce více konkrétní. Je popsáno, jak jsou událostně orientované aplikace konstruovány a jak uvnitř fungují. Jsou také rozebrány bloky, ze kterých se aplikace skládají. Postupně je opouštěno od abstrakce, až do poslední části této práce, ve které je popsána konkrétní událostně orientovaná aplikace.

3.4.1 Událostně procesní architektura

Každá aplikace je svým způsobem jedinečná, existuje ale jistá struktura, kterou mají událostně orientované aplikace společné.

Definice: ***Vydavatel událostí*** *je entita na pokraji událostně řídícího systému, která systému uvádí detekované události* (Etzion & Niblett, 2011).

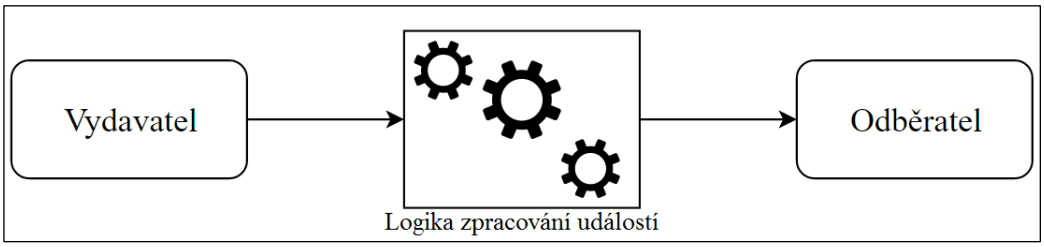
Aplikace obsahuje jednu nebo více komponent generující události. Jak již bylo popsáno v části o notifikacích *(viz Notifikace)*, není tím myšleno, že by samotné události byly komponentou vytvářeny. Znamená to, že detekuje jejich výskyt. Tato komponenta je v práci označována jako vydavatel událostí. Vydavatel se může vyskytovat v mnoha formách, například to mohou být hardwarové senzory, které produkují události v případě, že zpozorují fyzický výskyt události. Mohou být také v podobě softwaru produkujícího události například v momentě, kdy se vyskytne nějaká chyba.

Definice: ***Odběratel událostí*** *je entita přijímající události systému* (Etzion & Niblett, 2011).

Protějškem vydavatele událostí je odběratel událostí. Těmito komponentami jsou v podstatě přijímány a zpracovávány události. I u odběratelů se mohou jejich funkce měnit.

Definice: ***Událostně procesní zprostředkovatel*** *je softwarová komponenta zpracovávající události* (Etzion & Niblett, 2011).

Vydavatelé a odběratelé jsou propojeni událostně distribučními mechanismy a často také událostně řídícím zprostředkovatelem, který je mezi ně umístěn, jak je možno vidět na obr. 3.7. Vydavatel je spíše ve vztahu s událostmi, které produkuje, než ve vztahu s jejím odběratelem. Není si vědom toho, kolika odběrateli jsou přijímány jím vyprodukované události, ani neví, jaké činnosti jsou odběrateli prováděny. Stejně tak odběratel reaguje spíše na samotnou událost než na činnosti vydavatele (Etzion & Niblett, 2011).



Obrázek 3.5 Struktura událostně procesní aplikace

3.5 Událostně procesní síť

Jak již bylo zmíněno, jednou z vlastností událostního řízení je abstrakce. Událostně procesní síť je cestou, jak abstrakce dosáhnout a v následujících částech je podrobněji popsána. Popisovaný koncept je implementačně nezávislý a není vázán na žádný programovací jazyk. Implementace v konkrétním programovacím jazyce je popsána v poslední části této práce.

Definice: ***Událostně procesní síť*** *je soubor propojených událostně procesních zprostředkovatelů, vydavatelů a odběratelů událostí* (Etzion & Niblett, 2011).

3.5.1 Modelování událostně procesní sítě

V této části jsou popsány komponenty, ze kterých se událostně procesní síť skládá a je také vysvětleno, jak je možno tyto komponenty propojit.

Stavební bloky

Událostně procesní sítě se obecně skládají z těchto základních stavebních bloků:

- typ události,
- vydavatel událostí,
- odběratel událostí,
- událostně procesní zprostředkovatel,
- kanál.

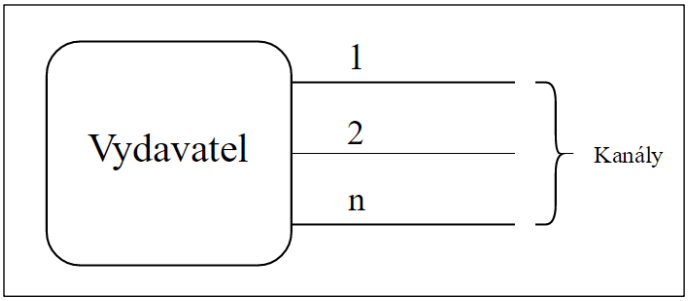
Ve složitějších aplikacích mohou být obsaženy další bloky, jako například kontext, nebo globální stav (Etzion & Niblett, 2011).

Událostně procesní stavební blok reprezentuje událostně procesní pojem, který je využit pro tvorbu platformě nezávislých definic prvků sítě. Pro ilustraci je možno použít příklad z chemie. Stavební bloky jsou jako chemické prvky, definice prvků sítě jako atomy a aplikace, které se z nich skládají jako molekuly (Etzion & Niblett, 2011).

Událostně procesní aplikace může obsahovat jeden nebo více typů události. Typ události je stavební blok, díky kterému je umožněno tyto typy popsat. Tímto blokem je definována struktura události, někdy označována jako schéma události. Vydavatel událostí je reprezentován aplikační entitou odesílající událostí do sítě a odběratel je entitou tyto události přijímající. Aplikace může obsahovat několik vydavatelů i odběratelů. Událostně procesní zprostředkovatel je blokem obsahující procesní logiku, který je umístěn mezi vydavatele a odběratele. Úkolem kanálu událostí je směřovat události mezi vydavateli a odběrateli. Všechny tyto bloky již byly v práci zmíněny a v jisté míře popsány (Etzion & Niblett, 2011).

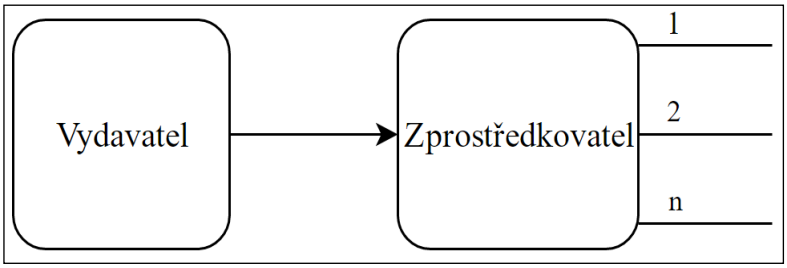
Kanál

Vydavatelem událostí mohou být události směrovány do libovolného počtu kanálů. Je v něm obsažena vnitřní logika mapující notifikace do kanálů. Vydavatelé mohou směřovat notifikace do kanálů na základě jejich typu, kanál však není omezen na přenos notifikací pouze jednoho typu (Fasion, 2006).



Obrázek 3.6 Kanály

Ve složitějších případech, ve kterých by nebylo vhodné, kdyby byla logika mapování notifikací obsažena přímo ve vydavateli, je možno využít zprostředkovatele této logiky (obr. 3.6). Vydavatel tak může jednoduše zasílat veškeré notifikace zprostředkovateli, ve kterém je logika obsažena. Zprostředkovatelem jsou nadále notifikace směřovány do kanálů (Etzion & Niblett, 2011).

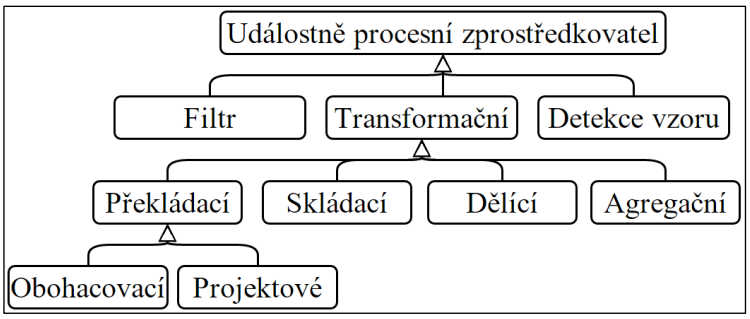


Obrázek 3.7 Využití zprostředkovatele

Událostně procesní zprostředkovatel

Existuje několik druhů událostně procesních zprostředkovatelů. Jsou vyobrazeny na obr. 3.8. Šipkou je znázorňován vztah dědičnosti[4].

[4] Dědičnost je proces, při kterém jeden objekt získá vlastnosti jiného objektu – svého předka. Jeho vlastnosti dále rozšiřuje (Schildt, 2014).



Obrázek 3.8 Druhy událostně procesních zprostředkovatelů

Filtr: Je používán pro eliminaci nepodstatných událostí. Příchozí události jsou přijaty filtrovacím zprostředkovatelem a následně podrobeny testu, kterým je rozhodováno, jestli mají být události vyřazeny, nebo předány k dalšímu zpracování. Příkladem může být test, který vyřadí veškeré události „zboží zakoupeno“, u nichž pořizovací částka nepřesáhla hodnotu 1000 korun.

Detekce vzoru: Těmito zprostředkovateli jsou přijímány kolekce událostí, nadále zkoumány za účelem nalézání vzorů událostí, které mohou být užity pro další členění.

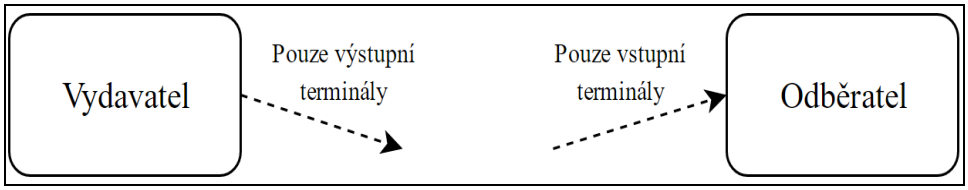
Transformační: Tímto typem je upravován obsah přijímaných událostí. Transformační mohou být dále na základě jejich vstupů a výstupů děleny na překládací, agregační, dělící a skládací. Překládací mohou být dále děleny na obohacovací a projektové. Obohacovacími jsou k přijatým událostem informace přidávány, projektovými jsou informace redukovány (Etzion & Niblett, 2011).

Již bylo zmíněno, jakou mohou mít zprostředkovatelé funkci. Nyní je krátce popsáno, jak mohou být zprostředkovatelé propojení s dalšími bloky, aby mohla být uskutečňována plnohodnotná interakce s bloky aplikace.

Obrázek 3.9 Událostně procesní zprostředkovatel

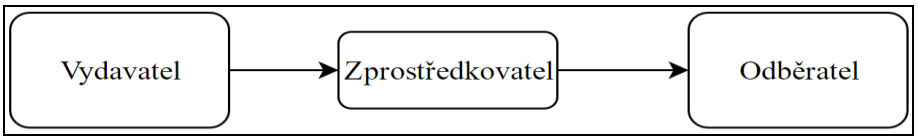
Na obr. 3.9 je zobrazen zprostředkovatel. Zprostředkovatel může mít jeden nebo více vstupních terminálů, kterými může přijímat zprávy od ostatních entit. Má také jeden nebo více výstupních terminálů, kterými vysílá události. Počet vstupních i výstupních terminálů se liší na základě toho, o jaký typ zprostředkovatele se jedná. Vstupní i výstupní terminál může mít předepsanou množinu typů událostí, které je

připraven přijímat a odesílat. Kromě toho také mohou být vstupním terminálům přiřazeny filtry, které vyřadí nevyžádaný typ událostí (Etzion & Niblett, 2011).



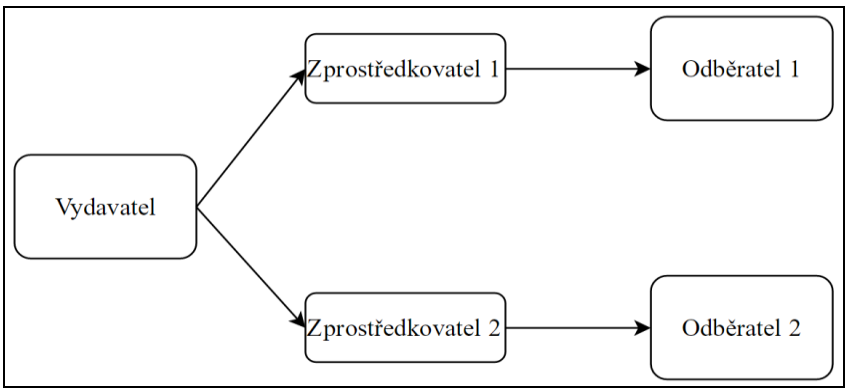
Obrázek 3.10 Vydavatel a odběratel

Vydavatelé a odběratelé jsou na tom podobně. Mají terminály, ale vydavatelé jen výstupní a odběratelé jen vstupní. Událostně procesní síť je vytvořena propojením vstupních a výstupních terminálů kolekce vydavatelů, odběratelů a zprostředkovatelů. Síť může být statická, obsahující fixní počet komponent, nebo může být dynamická. V případech, kdy je síť dynamická, mohou vydavatelé a odběratelé přibývat i ubývat dynamicky za chodu aplikace. Může být také měněn způsob, jakým jsou jednotlivé bloky vzájemně propojeny. Nyní je krátce popsáno, jak mohou být bloky propojeny (Etzion & Niblett, 2011).



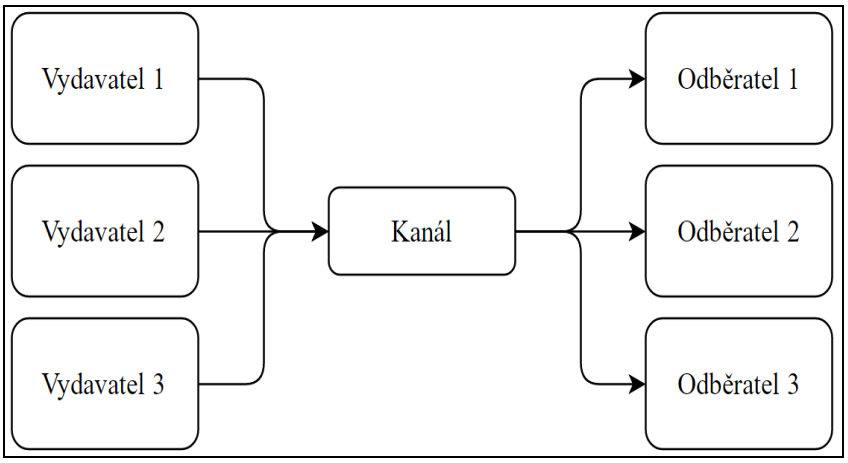
Obrázek 3.11 Jednoduchá událostně procesní síť

Na obr. 3.11 je vyobrazena jednoduchá událostně procesní síť, kde jsou bloky vydavatel, zprostředkovatel a odběratel zastoupeni jednou entitou. To ale neznamená, že je přítomen například jediný fyzický sensor pohybu jako vydavatel. Vydavatelem může být celá množina sensorů stejného typu (Etzion & Niblett, 2011).



Obrázek 3.12 Vydavatel s více výstupními terminály

Ve složitějších případech je potřeba zvolit už více sofistikované řešení. V takových situacích je vhodné využít stavebního bloku kanál (obr. 3.13).

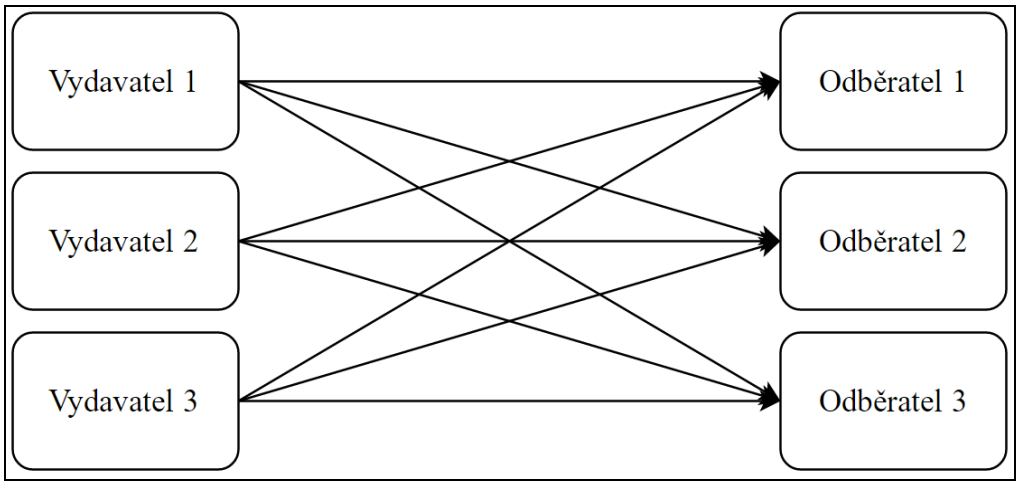


Obrázek 3.13 Užití kanálu k distribuci událostí

Z využití kanálu plynou jisté výhody:

- Kanály mají identifikátory, což znamená, že bloky vydavatelů, odběratelů i zprostředkovatelů mohou být navázány na kanál, a ne jeden na druhého. To také snižuje komplexnost diagramu, jak je možno vidět na obr. 3.13. Je pak také mnohem jednodušší přidat nebo odebrat jednotlivé bloky.
- Kanály mají své vlastní konfigurační parametry. Je tedy možno specifikovat konkrétní chování směřování událostí.

Jak je možno vidět na obr. 3.14, k dosažení stejné funkčnosti bez užití kanálů by každý odběratel musel být propojen s každým dodavatelem. Porovnáním obrázků (3.13, 3.14) je možno vypozorovat, jak je toto řešení složitější (Etzion & Niblett, 2011).



Obrázek 3.14 Složitost distribuce událostí bez užití kanálu

3.6 Shrnutí

Úvodem této kapitoly byly vysvětleny principy událostně řízeného programování. Byly popsány způsoby distribuce událostí a možné propojení komponent definující způsob interakce mezi těmito komponentami událostně procesních architektur. V poslední části byly definovány základní stavební bloky událostně orientovaných aplikací a možné způsoby jejich propojení.

Díky událostně řízeného programování je možno docílit toho, aby na sobě navrhované komponenty byly nezávislé. Z toho plynou různé výhody, například to, že dané komponenty mohou být využity opakovaně a jsou na sobě nezávislé. Dále bylo zjištěno, že spouštěním událostí díky asynchronnímu přístupu k interakci daných komponent je možno snížit procesní latenci zpracování událostí. Je však možné tento přístup kombinovat i s jinými typy interakce, díky čemuž může být efektivita v některých případech ještě více navýšena.

Dále je možno nastavit způsob komunikace jednotlivých komponent. Například užitím událostně procesního zprostředkovatele, jímž může být zastoupena funkce filtru, je možno docílit snížení počtu událostí, jež mají být zpracovány, jelikož ty nepodstatné jsou vyřazeny. Užitím kanálů může být také značně zjednodušena komplexnost událostně procesních sítí, což také navyšuje výkon a přehlednost.

4 Návrh a implementace aplikace

V této kapitole práce je demonstrováno, jak byly získané poznatky převedeny do praxe. Praktickým výstupem této části je událostně orientovaná aplikace. Aplikace byla vyvinuta v programovacím jazyce Java, kterému je věnována krátká část práce. Aplikace obsahuje událostně procesní síť.

4.1 Java

První vydání jazyka Java v roce 1996 vyvolalo obrovskou vlnu vzrušení nejen ve světe informačních technologií, ale také v hlavních médiích jako *New York Times, Washington Post* a *BusinessWeek* (Horstmann, 2016). Není pochyb, že Java je jedním z těch lepších programovacích jazyků dostupných seriózním programátorům. Mezi základní důvody, které tento jazyk činí tak dobrým jsou ty, že je: jednoduchý, objektově orientovaný, bezpečný, architektonicky nezávislý, portabilní, vysoce výkonný, dynamický a mnohé další (Horstmann, 2016). Práce s událostmi v jazyce Java je možná díky balíčku `java.awt`, nebo `javafx.event` (Horstmann, 2016).

4.2 Aplikace

K vývoji událostně procesní sítě ve tvořené aplikace nebylo využito tříd, které jazyk Java nabízí pro práci s událostmi. Je to z důvodů, že tyto třídy jsou určeny pro události přímo souvisejícími s grafickým uživatelským rozhraním. Tato aplikace zpracovává události v jiném kontextu. Při návrhu sítě byly využity poznatky nabyté při tvorbě práce. Úvodem jsou však nejdříve popsány drobnosti, které se práce týkají pouze nepřímo.

4.2.1 Jazyk zdrojového kódu

Samotný text práce, až na některé výjimky v názvech, je psán v českém jazyce, i když některé termíny po přeložení do českého jazyka zní poněkud uměle. Zdrojový kód aplikace byl už ale psán v jazyce anglickém.

V jazyce Java jsou definovány jmenovací konvence užívané pro *JavaBeans*. *JavaBeans* jsou znovupoužitelné softwarové komponenty. Tyto konvence souvisejí s takzvanými přístupovými metodami. Java je objektově orientovaný jazyk a jedním

z principů objektové paradigma je zapouzdřenost. Díky zapouzdřenosti je umožněno skrýt implementační detaily před uživatelem (Schildt, 2014). Zapouzdřenost je možno představit si jako ochranný obal, který chrání kód a data proti přístupu kódem, který je vně daný obal. Přístup k tomuto kódu je řízený jasně definovaným rozhraním. Toto rozhraní je tvořeno právě zmíněnými přístupovými metodami, kterými je umožněn přístup k jinak privátním atributům objektů. V těchto metodách může být také obsažena logika zabraňující nekorektnímu přístupu k datům uvnitř obalu. V případě potřeby nastavení hodnoty atributu je použita metoda, jejíž jméno je vytvořeno tím, že se před název atributu připojí slovo „*set*“ – nastavit. Před metodu sloužící k získání hodnoty je připojeno slovo „*get*“ – získat. U datového typu *boolean* je použito slovo „*is*“ (Boyarsky & Selikoff, 2015). Takže v konkrétním případě, jestliže by názvem atributu bylo česky psané například „jméno“, metody by se jmenovaly „getJméno“, „setJméno“. V případě, že by byl atribut typu *boolean*, třeba „mrtev“, metoda pro získání hodnoty atributu by nesla jméno „isMrtev“. Na první pohled je zcela viditelné, že míchat jmenovací konvence s jazykem českým je poněkud nepřirozené. Dalším důvodem je to, že samotné editory používají tyto konvence pro generování kódu, stejně tak jsou tyto konvence vyžadovány různými nadstavbovými technologiemi. Takže pojmenovat metodu „jeMrtev“ není řešení. Na vrchol toho všeho, stejně ošklivě to vypadá, když se česká slova míchají se slovy klíčovými, které jsou samozřejmě také v angličtině. Proto byl pro kód zvolen jazyk anglický. V následujících částech práce, kde je odkazováno na konkrétní část aplikace, je taky využito angličtiny.

4.2.2 GitHub

Při vývoji aplikace byl využíván systém správy verzí Git. Tímto systémem správy verzí je umožněno kdykoliv obnovit předchozí verze projektu. Nejen pro programátory, ale i pro grafiky a webdesignery je systém správy verzí ideálním nástrojem. Při jeho používání je možno vrátit jednotlivé soubory nebo i celý projekt do předchozího stavu, porovnávat změny a mnohé další (Chacon, 2009). Díky systému Git bylo ušetřeno spoustu trápení při obnovování předchozích verzí v momentech, kdy aktuální vývoj nebyl vyhovující.

4.2.3 Představení aplikace

Na začátku vývoje byl účelem aplikace pouze trénink pro řešení komplexnějších problémů a příprava pro budoucí zaměstnání. I přes tu nejasnou formulaci bylo jasné

aspoň to, že se jedná o primitivní hru žánru RPG (role-playing game), kde role hrdiny je zastoupena uživatelem. Neexistovala však vize nebo určitý cíl, ke kterému by aplikace měla směřovat. Aplikace pomalu nabývala na objemu a rozšiřovala se až do bodu, kdy bylo jasné, že je pro řešení nutno využít sofistikovanějších metod než obyčejné nestrukturované a neřízené psaní kódu. V momentě, kdy bylo napsáno přibližně pět set řádků kódu, bylo zřejmé, že je nutno vývoji poskytnout jistý cíl a průběžně definovat následující kroky. Milníkem daného okamžiku bylo docílit toho, aby na sobě jednotlivé komponenty byly co nejméně závislé. Hlavní motivací bylo to, že s neustále narůstajícím množstvím jednotlivých komponent roste také potřeba úprav a dolaďování. V případě, že by na sebe dané komponenty byly závislé, při sebemenší změně by bylo nutno upravit nejen samotnou část, které se změna přímo týká, ale také všechny ostatní části, které jsou na dané komponentě závislé. Samozřejmě nejde odstranit veškerou závislost, ale cílem bylo udržovat ji na minimu.

S nezávislostí a odděleností jsou spjaty vrstvy aplikace. Aplikace je tvořena třemi vrstvami – datovou, aplikační a prezenční. Data jsou uchována v souborech typu CSV *(comma-separated values).* Tímto způsobem je zajištěna jednoduchá práce s daty. U aplikací středního až velkého rozsahu je vhodnější mít data uchovaná v databázi. Jelikož se ale jedná o aplikaci menších rozměrů, po zvážení bylo rozhodnuto o vytvoření vlastního systému v souborech. Není to však rozhodnutí definitivní. Při budoucím rozvoji aplikace může být rozhodnuto o migraci datové vrstvy do databázového systému.

Již bylo zmíněno, jak je důležité zachovat oddělenost a nezávislost komponent. To se týká také aplikační vrstvy. Díky tomu, že je aplikace stavěna na tomto principu, bylo přidávání a úprava funkcionalit mnohem jednodušší. V části práce, ve které byly vysvětlovány událostně procesní platformy, bylo vysvětleno, že oddělenost je důležitým východiskem pro událostní řízení *(viz Událostně procesní platformy).* Díky tomu, že při vývoji bylo apelováno na to, aby byla oddělenost zachována, mohla být do aplikace zavedena událostně procesní síť tvořící významnou část aplikační vrstvy. Další podstatnou částí této vrstvy je model. V modelu je obsažena logika hrdiny samotného, světa, neutrálních postav, nepřátel atd. Hrdina i svět je ovládán událostmi vznikajícími buďto v závislosti na jistých okolnostech, nebo na základě akcí hráče. Možnost provést tyto akce jsou zprostředkovány třetí vrstvou – uživatelským rozhraním.

V následující části je popsána pro dnešní dobu mírně atypická část. Jedná se o způsob implementace uživatelského rozhraní. Hra je ovládána v prostředí příkazového řádku. Inspirací pro tvorbu hry v textovém formátu je film „*Velký“ (1988)*, kde byl do hlavní role obsazen *Tom Hanks*. Klíčovým prvkem pro tuto inspiraci je pouhý detail, kterému bylo ve filmu věnováno jen pár vteřin. V sedmi letech byl pro mě počítač nedosažitelným snem. A právě tento nedosažitelný sen byl vyobrazen hned v první scéně, kde malý *David Moscow* hraje počítačovou hru ovládanou právě textovými příkazy. Na to, co je dnes ve hrách nahrazeno propracovanou grafiko, se před několika lety dalo nahlížet jako na vrchol vědy a techniky. A přestože je možno na toto zpracování pohlížet jako na zastaralé, touha vyzkoušet takovou ve mne přetrvala. Z toho důvodu je hra zpracována právě v této podobě. Prezentační vrstva je navrhována s opatrností, aby jediná logika zahrnutá v této vrstvě s touto vrstvou přímo souvisela. Tím je umožněno tvořit ji jako nezávislou část, kterou by nebyl problém nahradit například právě grafickou verzí. Toto rozhraní je také prostředkem umožňujícím uživateli předkládat požadavky na to, jaké akce se mají provést. Dané požadavky jsou následně zpracovány událostně procesní sítí, jejíž principy byly dostatečně vysvětleny v teoretické části práce.

4.3 Stavební bloky aplikace

V předchozích částech práce byly popsány stavební bloky událostně procesních sítí. V následující části jsou rozebrány také stavební bloky, ale ne ty související se sítěmi, na ty přijde řada až později. Tato část je věnována stavebním blokům, ze kterých je tvořena celá aplikace.

4.3.1 Data a zdroje

Jak již bylo řečeno, pro uchování zdrojových dat není použita databáze, ale zdroje jsou uchovávány v programovém balíčku `resources`, který obsahuje datový model s CSV soubory, které jsou lehce upravitelné například v MS Excelu. Zdrojová data budou v práci dále nazývány jako zdroje.

```
merchant.weapons.blacksmith;10;5;5;5;;item.weapon.blacksmith_hammer;WARRIOR;humanoid
merchant.weapons.hunter;8;4;7;3;;item.merchandise.weapon.ranged.bow;RANGER;humanoid
merchant.weapons.wizard;7;6;10;1;;item.merchandise.weapon.wand.staff;SORCERER;humanoid
merchant.consumables.drink_shop.joe;4;8;2;1;;item.weapon.broken_bottle;WARRIOR;humanoid
merchant.consumables.food_shop.gordon;4;8;2;1;;item.weapon.dry_bread;WARRIOR;humanoid
innkeeper.tavern.gwen;5;9;3;2;;item.merchandise.weapon.melee.dagger;WARRIOR;humanoid
```

Zdrojový kód 1 Ukázka CSV souboru

Při inicializaci aplikace jsou zdroje načteny. Správné načtení a uložení jsou zprostředkovány třídou `ResourceCache`. V případě potřeby využití některého ze zdrojů je možno využít rozhraní `CloneInterface`.

```
package cz.vsb.ekf.lan0116.util.cloner;

public interface CloneInterface<T extends Object> {
    T clone(String objectId);
}
```

Zdrojový kód 2 Rozhraní CloneInterface

Jednotlivé implementace rozhraní následně obsahují podrobnosti, jak se má daný typ objektu klonovat. Metody ve třídách implementující toto rozhraní nevrací referenci na objekt uložený v `ResourceCache`, ale referenci na nově vytvořený objekt. Tím je zajištěna neměnnost objektů uložených v paměti.

```
public class ResourceCache {
    private final Map<String, Attack> attackMap;
    private final Map<String, Consumable> consumableMap;
    private final Map<String, Creature> creatureMap;
    private final Map<String, Merchandise> merchandiseMap;
    private final Map<String, Weapon> weaponMap;
...
}
```

Zdrojový kód 3 Třída ResourceCache[5]

4.3.2 Context

`Context`[6] je komponenta, jež zastupuje funkcionalitu podobnou aplikačnímu serveru. Je v něm držen aktuální stav hry, který je především určen stavem hrdiny

[5] V případě, že je v kódu uvedeno „…“, znamená to, že část kódu je vynechána. Tak mohou být ukázány pouze důležitější části a nemusí být uvedeno zbytečně velké množství kódu. Toto pravidlo je používáno pouze pro oddělení dvou částí kódu. Není užíváno například pro nahrazování vypisování balíčku, jelikož v případě, že je balíček uveden, musí být v souboru vždy na prvním místě, tudíž výrazu nepředchází žádný kód (Boyarsky & Selikoff, 2015). V případě vynechání balíčku v ukázce kódu je obvykle vynechán i seznam importů.

[6] Název atributu „`Context`“ byl inspirován kontextem používaným např. v Java EE aplikacích, kde je `context` aplikace držen kontejnery. V práci je pro tuto komponentu používán název „`Context`“ s „C“, aby byl odlišen od slova „kontext“, jehož významem je myšlena „souvislost“.

a světem. `Context` je model aplikace obsahující komponenty, kterými může být aktuální stav měněn. V proměnné `hero` je uložena reference na instanci hrdiny ovládaného hráčem. Svět je popsán podrobněji v této kapitole. Proměnná `cache` již byla popsána. Je zde uložen také `scanner`, aby se nemusela vytvářet nová instance pokaždé, když je potřeba `scanner` využít. Jelikož se jedná o hru ovládanou pomocí příkazového řádku, používá se poměrně často. `Localization` obsahuje hashovací tabulku, která je načtena ve fázi inicializace hry. Díky tomu je umožněno zvolit si jazyk aplikace. Na výběr je jazyk anglický a jazyk český[7]. Akce uživatele jsou řízené třídou `Session`. Atributy třídy `Session` budou podrobněji rozebrány v pozdější části kapitoly.

```
public class Context {
    private final Hero hero;
    private final World world;
    private final ResourceCache cache;
    private final Scanner scanner;
    private final Localization localization;
    private final Session session;
...
}
```

Zdrojový kód 4 Třída Context

4.3.3 Svět

Svět (`World`) je dalším atributem třídy `Context`. Všechny ostatní atributy byly již zmíněny. U některých, jako je třeba `ResourceCache`, byl uveden základní popis funkcionalit. Popisu hrdiny byla věnována již jen jedna věta. Cílem této práce není podrobně nebo aspoň stručně popsat každou část aplikace, pouze její důležité prvky. Jde především o popsání implementace událostně procesní sítě. Svět je však centrem veškerého dění, navíc je při jeho inicializaci využito zajímavých principů, proto je jeho popisu věnována rozsáhlejší část než popisu hrdiny.

Svět je vytvořen ve fázi inicializace hry. V inicializaci světa probíhá vytváření budov, jako jsou kupříkladu obchody. Aby mohla být vytvořena instance třídy `Shop`,

[7] Aplikace byla tvořena v anglickém jazyce, z důvodů již objasněných *(viz Jazyk zdrojového kódu)*. Stejně tak bylo původním záměrem tvořit samotné herní prostředí jen v jazyce anglickém. Ale jak již bylo řečeno úvodem této hlavní kapitoly, jednou z motivací pro tvorbu této aplikace bylo procvičit si programovací techniky. Z toho důvodu byla implementována možnost hrát hru i v českém jazyce. V momentě psaní této práce čeština není dokonalá, protože to není aktuálním cílem, jak hlavním, tak ani vedlejším. Dílčím cílem týkajícího se českého jazyka bylo umožnit si jej vybrat. Tento cíl, byť v poněkud kostrbaté podobě, byl splněn. Je však možné, že v budoucnu český jazyk bude dopilován na stejnou úroveň jazyka anglického.

musí být jako parametr konstruktoru předána reference na instanci třídy `Merchant` – obchodník. Nebylo by vhodné, aby se veškerá tato inicializace shlukovala na jednom místě. Ideální je, když třída má pouze jeden úkol, jednu zodpovědnost, tzv. „Princip jedné odpovědnosti“ (*Single Responsibility Principle*) (Joshi, 2016). S narůstající složitostí aplikace je těžší se tohoto principu držet. V Java EE aplikacích je k tomuto řešení užíváno tzv. „*dependency injection*“ (DI), kdy je možno jednotlivé úkoly delegovat na jiné komponenty (Coward, 2015). Aby mohlo být DI pohodlně užíváno, je vhodné, když aplikace funguje na aplikačním serveru, kterým je DI zprostředkována. Ve zpracované aplikaci není využito aplikačního serveru, tudíž není použito „ryzí“ DI, jak je obvykle implementováno v jazyce Java, ale funkcionalitu na tomto principu je poskytnuta právě třídou `Context`, díky kterému je možno řešit úkoly na jednotlivých místech. Instance tříd uložených jako atribut třídy `Context` je možno předávat tam, kde jsou zrovna potřebné. Úkolem třídy `Context` však není tyto instance vytvářet, ale pouze poskytovat k nim kontrolovaný přístup. Veškerá data, která souvisí s atributy třídy `Context` *(viz. Context)*, jsou dodána ve fázi inicializace.

```
public class Launcher {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in, "UTF-8");
//Volba lokalizace - jazyka hry
        System.out.println("Language/Jazyk");
        ... localization = new Localization(ResourceUtil
            .getResource(ResourceType.LOCALIZATION,"en"));
        ...

//Načtení potřebných dat do ResourceCache
        Map<String, Attack> attackMap =
            ResourceToMapUtil.createAttackMap(ResourceUtil
                    .getResource(ResourceType.ATTACK_ALL,
            "attacks"));
        ...
        ResourceCache cache = new ResourceCache(attackMap,
            consumableMap, creatureMap, null, weaponMap);

//poslední část inicializace třídy Context
        World world = ...
        Context context = new Context(hero, world, cache, scanner,
                localization, session);
        TextEvents textEvents = new TextEvents(context).playGame();
        }
}
```

Zdrojový kód 5 Třída Launcher

4.3.4 Uživatelské rozhraní

Funkcí uživatelského rozhraní je poskytnout uživateli prostředí pro hraní originálním způsobem. Rozhraním není možno zasahovat přímo do modelu, tudíž model nemůže být svévolně měněn uživatelem, jedině na základě přirozených akcí, které je uživatel schopen provést skrze uživatelské rozhraní. Rozhraní vidí na model, ale nemá k němu přímý přístup. To je ale dostačující k tomu, aby na model mohl uživatel reagovat. Je podstatné, aby model a uživatelské rozhraní byly oddělené. Tyto dvě části nesmí být propleteny, jinak by byl zhoršen další vývoj a zhoršila by se možnost rozšiřitelnosti.

4.4 Událostně řízený systém

V předchozí kapitole práce byly popsány distribuční modely událostně procesních sítí. Sítí implementovanou v aplikaci je využíváno kombinace modelu kanál a typ. Díky tomu je možno využít typu události k tomu, aby byly předávány k zpracování jen odběratelům, se kterým tento typ souvisí.

4.4.1 Session

Jak funguje interakce vzoru požadavek/odpověď a její kontext v událostně procesních systémech bylo vysvětleno v předchozí části práce *(viz Komunikace požadavek/odpověď v kontrastu s událostmi).* Na interakci uživatele s aplikací je možno nahlížet podobným způsobem. V případě, že se chce hráč například ozbrojit, vysloví požadavek, jakou zbraň by rád používal. Požadavek je následně zpracován událostně procesní sítí. Tyto události mohou skončit úspěchem, kdy je uživatel úspěšně vyzbrojen, nebo také neúspěchem. Neúspěchem může dopadnout třeba to, když je hráčem požadováno použít jako zbraň například chléb. Starým a ztvrdlým kusem chleba by umlátit nepřítele nejspíše možné bylo, ale ve hře chléb není součástí hierarchie třídy `Weapon`, proto tato událost skončí neúspěchem. Tento druh[8] událostí je definován rozhraním `Event`. Odlišný druh událostí je definován rozhraním `ServerEvent`. Ty se od interakce požadavek/odpověď liší, jelikož nemohou být

[8] Záměrně je použito slovo „druh“, protože slovo „typ“ je v tomto kontextu použito pro shlukování událostí podle tématu, kterého se týkají. Druhy jsou v tomto kontextu pouze dva: běžné události a serverové události. Rozdělení podle typů je sofistikovanější a je tomuto tématu věnována část práce.

vyvolány explicitně na přání uživatele. Podrobnější rozdíly společně s principy fungování událostí jsou rozebrány v následujících částech kapitoly.

4.4.2 Událostně procesní síť

Veškerá logika týkající se událostí je obsažena v programovém balíčku `eventSystem`, který je složen z dalších balíčků a tříd. V následující části je popsán obsah balíčku a to, jak je implementována událostně procesní síť.

Balíček events

V tomto balíčku jsou obsaženy třídy zabývající se jedním ze dvou hlavních druhů událostí, které byly popsány v úvodu této podkapitoly. Jde o události, které mohou být explicitně vyvolány uživatelem. Dále jsou zde obsaženy třídy zabývající se rozdělením událostí dle jejich typu a logikou těchto událostí. Události související s tímto programovým balíčkem jsou v práci označovány jako „běžné události". Veškeré třídy definující tento druh událostí implementují rozhraní `Event`. Pro rozlišení druhů události budou tento druh nadále v práci označován jako „běžná událost". Více o tom, jak je událost zpracována a jak může ovlivnit hru, je popsáno až po vysvětlení zbývajících komponent událosntě procesní sítě. Kdyby to bylo popsáno nyní, bez zbývajících znalostí o síti by tyto informace neměly takovou hodnotu, jakou budou mít tehdy, až budou vysvětleny v celkovém kontextu. Prozatím stačí vědět to, co již bylo o běžných událostech řečeno. V pozdější části je popsáno, jak drobná akce dovede uvést do pohybu celou událostně procesní síť.

```
package cz.vsb.ekf.lan0116.eventSystem.events;

public interface Event {

    EventType getType();

    default EventSuperType getSuperType() {
        return getType().getEventSuperType();
    }
}
```

Zdrojový kód 6 Rozhraní Event[9]

[9] Jak je možno z kódu vypozorovat, v rozhraní je metoda, která není abstraktní. Tato práce se nezabývá tím, co je třída, co rozhraní a co můžou a nemůžou obsahovat. Toto ale za zmínku stojí. Až do verze Java 8 metody rozhraní nemohli mít tělo, to ani prázdné. Ve verzi Java 8 byl představen nový typ metody rozhraní označovány jako „*default method*" (výchozí metoda) (Boyarsky & Selikoff, 2015). Před verzí 8 by za použití pouze rozhraní stejného efektu muselo být dosaženo tak, že by buď v každé třídě implementující toto rozhraní musela být deklarována tato metoda, přestože by měla vždy stejné tělo. Kdyby takových tříd bylo sto, znamenalo by to, že metoda musí být stokrát deklarována. Jelikož

V části práce zabývající se stavebními bloky událostně procesních sítí *(viz Stavební bloky)* bylo řečeno, že jedním z bloků je typ události. V momentě expanze aplikace bylo zjištěno, že dělení běžných událostí do typů jedné úrovně není dostačující, protože s každým novým typem běžné události, který může v budoucnu vzniknout, klesá přehlednost. Nepřehlednost a nízká čitelnost je při programování velice nežádaným jevem. Proto jsou typy událostí (`EventType`) shlukovány do skupin (`EventSuperType`). Díky metodám obsažených v rozhraní je možno zjistit, jakého je událost typu a do jaké skupiny se řadí.

```
package cz.vsb.ekf.lan0116.eventSystem.events;

public enum EventSuperType {
    COMBAT,
    GAME,
    HERO,;
}
```

Zdrojový kód 7 Výčet EventSuperType

V aplikaci jsou definovány tři skupiny událostí. K deklaraci těchto skupin je použito výčtového datového typu `enum`. Skupina *`COMBAT`* (`AttackMoveEvent`, `HealEvent`, `FleeEvent`) je složena z událostí souvisejících s bojem, skupina *`GAME`* se hrou (`NewGameEvent`, `GiveUpEvent`) a skupina *`HERO`* s hrdinou (`ConsumeEvent`, `EquipEvent`). Samozřejmě jako příklady nejsou jmenovány všechny události, pouze některé pro ilustraci.

```
package cz.vsb.ekf.lan0116.eventSystem.events;

public interface EventType {
    EventSuperType getEventSuperType();
}
```

Zdrojový kód 8 Rozhraní EventType

Typ událostí je definován rozhraním `EventType`. V rozhraní je obsažena metoda `getEventSuperType`. Ta je v jednotlivých implementacích deklarována na základě toho, pod jakou skupinu událost spadá. Skupiny a typy jsou linkovány sofistikovaným způsobem, který je vysvětlen v následující části.

jsou programátoři líná stvoření, není to moc ideální řešení. Od verze Java 8 však může být metoda označena klíčovým slovem „default“ a může obsahovat tělo. Kód bude bez problému zkompilován a metoda bude děděna všemi třídami implementující toto rozhraní.

```
public enum HeroType implements EventType {
    CONSUME,
    DROP,
    EQUIP,
    ...
    TRADE,
    TRAVEL,;

@Override
public EventSuperType getEventSuperType() {
    return EventSuperType.HERO;

}
```

Zdrojový kód 9 Výčet HeroType

Události jsou definovány třídou implementující rozhraní `Event`. S každou takovou událostí je svázána hodnota výčtu implementující rozhraní `EventType`. Konkrétní výčty jsou: `CombatType`, `GameType`, `HeroType`. Výčtem je přepsána metoda `getEventSuperType`. Touto metodou může být zjištěna konkrétní skupina, do které se typ události řadí, takže u každé události je možno zjistit její typ i skupinu. Pro připomenutí: v rozhraní *Event* je obsažena metoda `getSuperType` a `getEventType`. V případě volání metody `getSuperType` je vrácena hodnota výčtu `EventSuperType`, kterou je určeno, do které skupiny typ události patří. Voláním metody `getType` je pak vrácena konkrétní hodnota výčtu této skupiny, kterou je určen konkrétní typ události, jak je možno vidět v zdrojovém kódu 10.

```
public class ConsumeEvent implements Event {

    private final Consumable subjectOfConsumption;

    public ConsumeEvent(Consumable subjectOfConsumption) {
        this.subjectOfConsumption = subjectOfConsumption;
    }

    public Consumable getSubjectOfConsumption() {
        return subjectOfConsumption;
    }

    @Override
    public EventType getType() {
        return HeroType.CONSUME;
    }
}
```

Zdrojový kód 10 Třída ConsumeEvent

Balíček failures

Jak již bylo naznačeno, není zaručeno, že každá běžná událost dopadne úspěchem. Pro vypořádání se s takovými situacemi je poskytnut obsah balíčku

failures. Pro informování uživatele o výsledku běžné události je zavedena třída Response.

```
public class Response {

    public static final Response SUCCESS = new Response();
    private final boolean success;
    private final FailureCause failureCause;

    ...

    public Response(FailureCause failureCause) {
        ...
    }

    ...

    public FailureCause getFailureCause() {
        return failureCause;
    }
}
```

Zdrojový kód 11 Třída Response

Po zpracování události jejím odběratelem je vrácena uživateli instance třídy Response se zprávou buďto o úspěchu konstantou *SUCCESS*, nebo je použito konstruktoru, jehož parametrem je rozhraní FailureCause obsahující metodu getEventType. Možná selhání jsou definována ve výčtech implementujících toto rozhraní. Každý typ události, u kterého může dojít k selhání, má svůj specifický výčet.

```
public enum CombatFailure implements FailureCause {
    ENEMY_DEAD,
    FLEE_DISABLED,
    FLEE_WEAK,
    NOT_ENOUGH_STAMINA,
    TARGET_DEAD,
    UNKNOWN,
    YOU_DIED,
    /**/;

    @Override
    public EventType getEventType() {
        return CombatType.ATTACK_MOVE;
    }
}
```

Zdrojový kód 12 Výčet CombatFailure

Balíček serverEvents

V této práci události označované jako serverové fungují jiným způsobem než běžné události. Pro ilustraci tohoto rozdílu je uveden následující příklad. V případě, že se člověku vybíjí mobilní telefon, jeho přirozenou reakcí je připojení jej

k nabíjecímu kabelu. Akcí vlastníka telefonu je tedy „připojení telefonu do sítě“. Toto je z pohledu této práce běžná událost. Vlastník si je vědom, jak danou akci provést. Odpovědí zprostředkovanou třídou `Response` by v tomto případě byla indikace na display, že se baterka úspěšně nabíjí. To, co již uživatel neřeší, je to, jak je nabíjení implementováno. Nezajímá ho, jak je střídavý proud díky diodovému usměrňovači měněn na stejnosměrný a umožňuje tak úspěšné nabití tohoto zařízení. A právě to „nabíjení“ je v kontextu této práce serverovou událostí. Pro ujasnění je ještě uveden konkrétní příklad ze hry. Hráč se rozhodne udeřit protivníka, tak provede úder – běžná událost. Protivníkovi v důsledku úderu ubydou pomyslné životy – serverová událost. V momentální podobě je ve hře implementován hlavně boj a obchodování. Serverové události zatím souvisí především s bojem.

Již bylo vysvětleno, že nejsou uživatelem vyvolány přímo, ale přesto se o jejich výsledku uživatel nějak dozvědět musí, stejně tak, jako je možné zjistit, zdali je telefon již plně nabitý, nebo není. Pro poskytnutí informací o boji je definováno rozhraní `FightResponse` obsahující výčet možných typů serverových událostí s bojem souvisejících. Tento typ je založen na podobném principu jako je typ u běžných událostí. Hodnoty výčtu jsou propojeny s třídami implementující rozhraní `FightResponse`.

```
package cz.vsb.ekf.lan0116.eventSystem.serverEvents.combat;

public interface FightResponse {

    FightResponseType getType();

    enum FightResponseType {
        DAMAGE_INFLICTION,
        STAMINA_CONSUMPTION,
        HEALING,
        ...;
    }
}
```

Zdrojový kód 13 Rozhraní FightResponse

Výskyty těchto událostí jsou sbírány voláním konstruktoru třídy implementující rozhraní `ServerEvent` – `BattleLogServerEvent`. Jak je možné vypozorovat z názvu, jedná se o log událostí souvisejících s bojem.

```java
public class BattleLogServerEvent implements ServerEvent {
    List<FightResponse> battleLog;

    public BattleLogServerEvent(List<FightResponse> battleLog) {
        this.battleLog = battleLog;
    }

    @Override
    public ServerEventType getType() {
        return ServerEventType.BATTLE_LOG;
    }

    public List<FightResponse> getBattleLog() {
        return battleLog;
    }
}
```

Zdrojový kód 14 Třída BattleLogServerEvent

Balíček eventProcessingNetwork

Tento balíček je poslední a velice důležitou součástí událostního systému aplikace. Po popisu tohoto balíčku je možno vysvětlit funkčnost celé sítě. Balíček obsahuje podbalíčky a třídy. V prvním balíčku `eventProcessingNetwork` se nachází rozhraní `EventSubscriber`, třída `EventPublisher` a abstraktní třída `EventHandler`. Rozhraní `EventSubscriber` (zdrojový kód 15) je implementováno třídou `EventHandler`, která je předkem pro třídy definující kanály (zdrojový kód 16). V současné době jsou v aplikaci obsaženy tři kanály. Třídou `EventPublisher` jsou události směřovány do patřičných kanálů (zdrojový kód 17). Událost je poté zpracována daným kanálem na základě jejího konkrétního typu.

```java
package cz.vsb.ekf.lan0116.eventSystem.eventProcessingNetwork;

import cz.vsb.ekf.lan0116.eventSystem.Response;
import cz.vsb.ekf.lan0116.eventSystem.events.Event;

public interface EventSubscriber {

    Response handleEvent(Event event);

}
```

Zdrojový kód 15 Rozhraní EventSubscriber

```
public class HeroChannel extends EventHandler {

    ...

    @Override
    public Response handleEvent(Event event) {
        HeroType eventType = (HeroType) event.getType();
        switch (eventType) {
            case CONSUME:
                ...
}
```

Zdrojový kód 16 Třída HeroChannel, potomek třídy EventHandler

```
public class EventPublisher {

    private final CombatChannel combatChannel;
    private final GameChannel gameChannel;
    private final HeroChannel heroChannel;

    ...

    private Response channelize(Event event) {
        switch (event.getSuperType()) {
            case COMBAT:
                return this.combatChannel.handleEvent(event);
            case GAME:
                return this.gameChannel.handleEvent(event);
            case HERO:
                return this.heroChannel.handleEvent(event);
            default:
                throw new UnsupportedOperationException("Channel " +
event.getSuperType() + " is not supported.");
        }
    }
}
```

Zdrojový kód 17 Třída EventPublisher

4.4.3 Princip funkčnosti událostně procesní sítě

Veškeré informace potřebné k pochopení principu fungování událostně procesní sítě aplikace byly již popsány, nyní tedy může být následující část věnována popisu funkčnosti sítě v celé její kráse. Z toho, co bylo doposud popsáno, by již mělo být zřejmé, jaká je podstata jednotlivých komponent figurujících v událostním procesu.

Ukázka boje

Je dána situace: hráč se nachází v boji. Aktuální hodnota atributu `contextu` výčtového typu `heroStatus` – hrdinův stav je „`HeroStatus.IN_COMBAT`“. Díky znalosti statusu hrdiny je zajištěno vykreslování správné obrazovky. Tato logika je logikou obstaranou uživatelským rozhraním, která však v této práci není více rozebrána, protože by to bylo přílišné odklonění od tématu. Stačí vědět, že díky správně vykreslenému rozhraní je hráči v takové situaci umožněno zvolit jednu z několika akcí, jak je možno vidět v ukázce výstupu (zdrojový kód 18). Kdyby se v daném boji potřeboval posílit, není problém, aby zkonzumoval nějaký nápoj či pokrm, který vlastní ve svém inventáři. Pro dodání trochu adrenalinu této části, je předpokládáno, že hráč zvolil agresivní akci a zaútočil. Na základě toho, jakou zbraní je hráč aktuálně vyzbrojen, jsou nabídnuty možnosti útoků. Některé zbraně mají k výběru více než jeden útok. Naneštěstí, v konkrétním příkladu se hráč zbraní nevyzbrojil, takže nemá jinou možnost než bít se holýma rukama. Jediné hráčovo štěstí je to, že nečelí zdatnému rytíři, ale pouze pouličnímu bídákovi. Pro demonstraci událostně procesní sítě však není nutná krvavá bitva.

```
Proti tobě stojí pouliční bídák
 0 Útoky                    | 1 Inventář                | 2 Prchnout
0

Vybrat útok:
 0 rána                     | 1 slabý úder              | 2 Zavřít
1

Poseroutka použil 0.0 energie
Poseroutka použil slabý úder jako útok
Poseroutka udělil 0.5 poškození do: pouliční bídák
pouliční bídák použil 0.0 energie
pouliční bídák použil slabý úder jako útok
pouliční bídák udělil 1.1 poškození do: Poseroutka
Proti tobě stojí pouliční bídák
```

Zdrojový kód 18 Ukázka výstupu

Jak je možno vidět (zdrojový kód 18), hráč skutečně vybral možnost „Útoky“. Po zvolení této možnosti je vyobrazen přehled dostupných útoků. V momentě, kdy se rozhodl, že uštědří „slabý úder“, vzniká událost `AttackMoveEvent`. Na úryvku kódu (zdrojový kód 19) je možno vidět, že je událost „vystřelena“ stylem nastíněným v předcházejících částech práce *(viz Spouštění událost)*. Je možno vidět, že volání události je ukládáno do proměnné typu `Response` (odpověď), přestože bylo řečeno, že při užití spouštění událostí není očekávána odpověď. Toto je vhodná příležitost pro vysvětlení toho, jak je komunikace událostí implementována. Odpověď opravdu očekávána není, tak jak to bylo vysvětleno v teoretické části. Ale není očekávána vydavatelem událostí. Tím je událost přesměrována do správného kanálu a již ho nezajímá, ani nemá možnost dozvědět se to, jak událost dopadla. Odpověď je určena pro uživatele. Po hlubším zkoumání kódu je ale možno vypozorovat, že z odpovědi je možno dozvědět se pouze to, jestli s úderem uspěl, nebo následkem nepřítelova protiútoku umřel. V ukázce výstupu je ale vidět, že se uživateli dostalo mnohem bohatější odpovědi. Tato odpověď je zajištěna díky existenci již popsaných serverových událostí, konkrétně `BattleLogServerEvent`.

```
Attack attack = this.getHero().getAttacks().get(tempChoiceNumber);
Response response = this.getContext().getSession().fireEvent(new
AttackMoveEvent(attack));

if (response.getFailureCause() != null) {
    if (response.getFailureCause().equals(CombatFailure.YOU_DIED)) {
        getContext().getSession().fireEvent(newRespawnEvent());
   }
}
```

Zdrojový kód 19 Vyvolání události Attack (útok)

Jak již bylo vysvětleno, serverové události nemohou být vyvolány uživatelem přímo. Není prostě možné, aby se jen tak rozhodl, že uštědří protivníkovi poškození, stejně tak, jako není možné někoho zabít pouhou myšlenkou. Tak, jako musí být vrahem vystřeleno ze zbraně, je třeba hráčem provést úder. Vystřelená událost reprezentující úder je zpracována třídou `CombatChannel` (zdrojový kód 20). Touto událostí bylo uvedeno do pohybu spoustu akcí. Akce jsou korigovány třídou `FightUtils`, která obsahuje metodu `attack`, jejíž úkolem je zpracování útoku (zdrojový kód 21). Metoda je zavolána v rámci zpracovávání události vyvolané hráčem.

```
List<FightResponse> battleLog = new ArrayList<>();
AttackMoveEvent moveEvent = (AttackMoveEvent) rawEvent;
Attack heroAttack = moveEvent.getAttack();
Creature enemy = hero.getHeroInteraction().getCurrentEnemy();
battleLog.addAll(attack(hero, heroAttack, enemy));

if (enemy.isAlive()) {
    battleLog.addAll(attack(enemy, selectAttack(enemy), hero));
} else {
    if (hero.getHeroInteraction().getEnemyQueue().isEmpty()) {
getHeroInteraction().setStatus(HeroInteraction.HeroStatus.READY);
    }

    getHeroInteraction().getEnemyQueue().remove();

    if (hero.getHeroInteraction().getEnemyQueue().isEmpty()) {
getHeroInteraction().setStatus(HeroInteraction.HeroStatus.READY);
    }
}
```

Zdrojový kód 20 Zpracování události Attack (útok)

```
public class FightUtils {
    ...

    public static List<FightResponse> attack(Creature attacker,
Attack attack, Creature defender) {
        List<FightResponse> battleLog = new ArrayList<>();

        float staminaCost = attack.getStaminaConsumption();
        if (attacker.getCurrentStamina() >= staminaCost) {
            attacker.decreaseCurrentStamina(staminaCost);
            battleLog.add(new StaminaConsumption(attacker,
staminaCost));

            float damageDealt = calculateDamage(powerOf(attacker,
attack), defender.getDefense(), attack.getPenetration());
            defender.decreaseCurrentLifeEssence(damageDealt);
            battleLog.add(new DamageInfliction(attacker, attack
,defender, damageDealt));

            ...

            if (!defender.isAlive()) {
                battleLog.add(new Information(defender,
Information.Info.DEATH));
            }
        } else {
            battleLog.add(new Information(attacker,
Information.Info.INSUFFICIENT_STAMINA));
        }
        return battleLog;
    }

    private static float powerOf...
    private static float calculateDamage...

    public static Attack selectAttack...
}
```

Zdrojový kód 21 Třída FightUtils

Metoda je volána nejdříve pro zpracování úderu hráče a v případě, že protivník hráčův útok přežil, je opět volána pro agresivní reakci hráčova aktuálního protivníka. Metodou je prvně zkontrolováno, jestli má vůbec hráč na útok dostatečné množství speciální energie, v kódu označovanou jako `stamina`. Tato energie je potřebná k provedení speciálních útoků, ale jelikož na pouhém úderu nic speciálního není, nevyžaduje speciální energii a může být bez problému zpracován dál. Událostně procesním systémem je zaznamenáno, kolik bylo spotřebováno hráčovi energie, v tomto případě žádná. Dále je vypočítáno poškození způsobené zvoleným útokem. Toto číslo je ovlivňováno i jinými faktory, jako například úrovní síly útočníka, výší obrany napadeného atd. Z toho důvodu tato čísla mohou být dosti proměnlivá. Když protivník hráčův útok přežil, nic mu nebrání provést protiútok. Ten je zpracován, stejně útok hráče. Stejně tak nepřítel může použít speciální útok, proto hráč předem nikdy neví, jak protiútok vůči němu dopadne.

Po veškerém tomto dění je list reprezentující bitevní log úspěšně zaplněn záznamy z bitvy. Ale jak již bylo několikrát řečeno, tyto události nebyly uživatelem vyvolány uživatelem a ze stejného důvodu si nemůže jen tak svévolně zažádat o informace o jejich výsledcích.

```
public class TextEvents {
     ...

  public void playGame() {
       ...

  //Nekonečná smyčka, možno ukončit událostí FleeEvent
    while (true) {
    //Kontrola, jestli se neodehrály serverové události
       ServerEvent serverEvent;
       while ((serverEvent =
   context.getSession().getResponses().poll()) != null) {
           if (serverEvent.getType() ==
           ServerEvent.ServerEventType.BATTLE_LOG) {
               BattleLogServerEvent event = (BattleLogServerEvent)
           serverEvent;
           //Získání případných serverových událostí
               List<FightResponse> battleLog = event.getBattleLog();
           //Vypsání veškerých událostí, které se za dané kolo
           odehrály
               for (FightResponse response : battleLog) {
                   this.printResponse(response);
               }
           }
```

Zdrojový kód 22 Třída TextEvents

Správné vypisování serverových událostí zaznamenaných v bitevním logu je obstaráno třídou `TextEvents`. Uživatelské rozhraní hry je vykreslováno v nekonečném cyklu, ten však může být ukončen událostí `FleeEvent`, kterou je umožněno hru explicitně ukončit. Na začátku každého cyklu je zjištěno, jestli se v průběhu hry vyskytly nějaké serverové události. K zpracování těchto událostí je využito principu popsaného v části pojednávající o interakci založené na principu požadavek/odpověď *(viz Distribuce událostí stylem požadavek/odpověď).* Následně voláním metody `printResponse` je zajištěno korektní vypsání serverových událostí. Tímto je ukončen proces zpracování události `AttackMoveEvent` vyvolanou akcí „slabý úder“, kterou bylo toto vše způsobeno.

4.5 Shrnutí

Tímto byly popsány základní principy fungování událostně procesní sítě. Tato kapitola byla věnována seznámení s aplikací a jejím hlavním konceptem. V převažující části byly popsány stavební bloky implementované událostně procesní sítě a byly také demonstrovány její funkcionality, jež vychází ze znalostí získaných při tvorbě teoretické části práce.

5 Závěr

Hlavním cílem této práce bylo analyzovat současné techniky návrhů událostně orientovaných architektur a využití těchto poznatků při návrhu a implementaci událostně procesní sítě vlastní aplikace.

Druhá kapitola práce byla věnována úvodu do událostního řízení. Byly objasněny události vyskytující se v každodenním životě. Událostmi je ve své podstatě formován běžný život člověka, některými výrazněji, jinými slaběji. A stejně tak mohou být formovány například různé aplikace, nebo informační systémy. Cílem této kapitoly bylo poskytnout základní potřebné informace potřebné pro následující část práce, která je věnovaná podrobnějšímu popisu událostního řízení. Cíl této kapitoly byl splněn.

Cílem třetí kapitoly bylo analyzovat současné techniky tvorby událostně procesních sítí. Byly podrobněji rozebrány událostně procesní architektury a principy, na kterých fungují. Bylo vysvětleno, jak hlavní komponenty událostně procesních architektur vzájemně komunikují a jak společně tvoří událostně procesní síť. Bylo zjištěno, že je možno využít různého propojení komponent sítě, stejně tak je možno zvolit různé způsoby interakce těchto komponent. Způsoby a možnosti využití těchto technik jsou součástí této kapitoly. Rovněž cíl této kapitoly se podařilo naplnit.

Čtvrtá kapitola této práce je založena na praktické části. Tou je událostně orientovaná aplikace. Začátkem bylo popsáno, o jakou aplikaci se jedná. Komponenty, které byly popsány v předcházejících kapitolách, byly předvedeny již v konkrétní podobě. Na základě analýzy v předchozí kapitole byly vybrány techniky, které by se nejvíce hodily pro tvorbu konkrétní událostně procesní sítě. Závěrem bylo ukázáno to, jak jedna zdánlivě drobná událost dovede uvést do pohybu celou síť. Demonstrování této funkcionality sítě bylo také cílem této kapitoly. I tento cíl byl splněn.

Vytvořená aplikace je výstupní částí práce. Využívá principů událostního řízení k dosažení efektivního běhu. Aplikace obsahuje funkční událostně procesní síť, která dovede zpracovat různé druhy i typy událostí. Aplikace je oddělena do na sobě nezávislých vrstev. Díky tomu je zajištěna možnost opakované použitelnosti jednotlivých komponent. Celá aplikace funguje dle předpokladů. Hlavní cíl byl s hrdostí splněn.

Funkcionalit aplikace je samozřejmě více než ta zmíněná. Jsou rozsáhlejší a komplexnější. Jejich popisu by mohlo být věnováno mnohem více stránek. V době tvorby této práce mají jen samotné zdrojové třídy Java více než 4500 řádků s kódem, dokumentací a komentáři. Událostně procesní síť obsahuje 1300 řádků. Je možno vidět, že v aplikaci se toho skrývá mnohem více. A to do toho nejsou započítány ostatní soubory, kterých není zrovna málo. Rozsah této práce je však značně omezen, proto zde nemůže být popsáno úplně vše. Tento důvod společně se skutečností popsanou v následujícím odstavci mě motivuje v práci v budoucnu nadále pokračovat. V praktické části s absolutní jistotou vím, že to nekončí. A co se týče části teoretické, možná se jednou vyvine na práci diplomovou.

Tato kapitola je vyvrcholením mé práce. Když nad tím tak přemýšlím, uvědomuji si to čím dál více. Jen této písemné části jsem věnoval nemalé množství času a byly momenty, kdy jsem úsilí věnované práci chtěl věnovat raději něčemu jinému. A nyní, když vím, že s touto prací končím, mi začíná být poněkud úzko. Když člověk čte výtečnou knihu je napjatý, jak to vše na konci dopadne. A už v momentě, kdy čte poslední stránku, mu začíná být po knize smutno. Závěr, na který se tak těšil, by nejraději odložil o několik stránek dál. A podobným pocitem jsem nyní naplněn i já. Čistě z osobního hlediska vnímám, že jsem tvorbou tohoto projektu získal ohromné množství zkušeností.

Seznam použité literatury

ALVES, A., SMITH, R. J. a L. WILLIAMS. *Getting Started with Oracle Event Processing.* 11. vyd. Birmingham: Packt Publishing Ltd., 2013. ISBN 978-1-84968-454-5.

AMNAJMONGKOL, Supannee. *Business Activity Monitoring with WebSphere Business Monitor.* 6. vyd.. New York: IBM International Technical Support Organization, 2008. ISBN 0738431133.

BABAEI, Z., RAHMANI, A.M. a REZAEI, A., 2016. Real-time reusable event-driven architecture for context aware systems. *Iranian Conference on Electrical Engineering, ICEE.* 2016, roč. 24, s. 294-299. ISBN: 978-1-4673-8789-7.

BHATT, C., DEY, N. a A. S. ASHOUR. *Internet of Things and Big Data Technologies for Next Generation Healthcare.* Cham: Springer International Publishing, 2017. ISBN 978-3-319-49735-8.

BINILDAS, C. A., BARAI, M. a V. CASELLI. *Service Oriented Architecture with Java.* Birmingham: Packt Publishing Ltd., 2008. ISBN 978-1-847193-21-6.

BOYARSKY, Jeanne a Scott SELIKOFF. *OCA: Oracle Certifi ed Associate Java SE 8 Programmer I Study Guide.* Indianapolis: John Wiley & Sons, Inc., 2015. ISBN: 978-1-118-95740-0.

BRUNS, Ralf a Jürgen DUNKEL. Towards pattern-based architectures for event processing systems. *Software—Practice & Experience*, 2014, 11. vyd., č. 44, s. 1395-1416. ISSN: 0038-0644.

COWARD, Danny. *Java EE 7: The Big Picture.* New York: McGraw-Hill Education, 2015. ISBN: 978-0-07-183733-0.

ETZION, Opher a Peter NIBLETT. *Event Processing in Action.* Greenwich: Manning Publications Co., 2011. ISBN 978-1-935182-21-4.

FASION, Ted. *Event-Based Programming: Taking Events to the Limit.* New York: Springer-Verlag New York, Inc., 2006. ISBN-10: 1-59059-643-9.

HOHPE, Gregor. Your coffee shop doesn't use two-phase commit. *IEEE Software*, 2005, 2. vyd, č. 22, s. 64-66. ISSN: 0740-7459.

HOHPE, Gregor a Bobby WOOLF. *Enterprise Integration Patterns : Designing, Building, and Deploying Messaging Solutions.* Boston: Addison-Wesley, 2005. ISBN 0-321-20068-3.

HORSTMANN, Cay. S. *Core Java.* 10. vyd. New York: Prentice Hall, 2016. ISBN-10: 0-13-417730-4.

CHACON, Scott. *Pro Git.* Praha: CZ.NIC, 2009, s. 17. ISBN: 978-80-904248-1-4.

JOSHI, Bipin. *Beginning SOLID Principles and Design Patterns for ASP.NET Developers.* Thane: Apress, 2016. ISBN-13: 978-1-4842-1847-1.

LUCKHAM, David. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Boston: Addison-Wesley Longman Publishing Co., 2001. ISBN:0201727897.

LUCKHAM, David. Foreword. *Event Processing in Action.* Greenwich: Manning Publications Co., 2011, s. xv–xvi. ISBN 978-1-935182-21-4.

LUCKHAM, David. *Event Processing for Business: Organizing the Real-Time Enterprise.* Hoboken: John Wiley & Sons, Inc., 2012. ISBN: 978-0-470-53485-4.

MIŚKOWICZ, Marek. Event-BasedControl: IntroductionandSurvey. *Event-Based Control and Signal Processing.* Boca Raton: CRC Press, 2015. ISBN-13: 978-1-4822-5656-7.

PASCHKE, Adrian. 2009. A semantic design pattern language for complex event processing. *AAAI Spring Symposium - Technical Report*, 2009, s. 54-60.

SCHILDT, Herbert. *Java The Complete Reference.* 9. vys. New York: McGraw-Hill Education, 2014. ISBN: 978-0-07-180856-9.

SCHMUTZ, G., LIEBHART, D. a P. WELKENBACH. *Service-Oriented Architecture: An Integration Blueprint.* Birmingham: Packt Publishing Ltd., 2010. ISBN 978-1-849681-04-9.

WESKE, Mathias. *Business Process Management.* Potsdam: © Springer-Verlag Berlin Heidelberg, 2007. ISBN 978-3-540-73521-2.

Seznam obrázků

Seznam zdrojových kódů

Prohlášení o využití výsledků diplomové (bakalářské) práce

Prohlašuji, že

- jsem byl(a) seznámen(a) s tím, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. – autorský zákon, zejména § 35 – užití díla v rámci občanských a náboženských obřadů, v rámci školních představení a užití díla školního a § 60 – školní dílo;
- beru na vědomí, že Vysoká škola báňská – Technická univerzita Ostrava (dále jen VŠB-TUO) má právo nevýdělečně, ke své vnitřní potřebě, bakalářskou práci užít (§ 35 odst. 3);
- souhlasím s tím, že bakalářská práce bude v elektronické podobě archivována v Ústřední knihovně VŠB-TUO a jeden výtisk bude uložen u vedoucího bakalářské práce. Souhlasím s tím, že bibliografické údaje o bakalářské práci budou zveřejněny v informačním systému VŠB-TUO;
- bylo sjednáno, že s VŠB-TUO, v případě zájmu z její strany, uzavřu licenční smlouvu s oprávněním užít dílo v rozsahu § 12 odst. 4 autorského zákona;
- bylo sjednáno, že užít své dílo, bakalářskou práci, nebo poskytnout licenci k jejímu využití mohu jen se souhlasem VŠB-TUO, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly VŠB-TUO na vytvoření díla vynaloženy (až do jejich skutečné výše).

V Ostravě dne 5.5. 2017

Vítězslav Lanc